University of Colorado
Colorado Springs

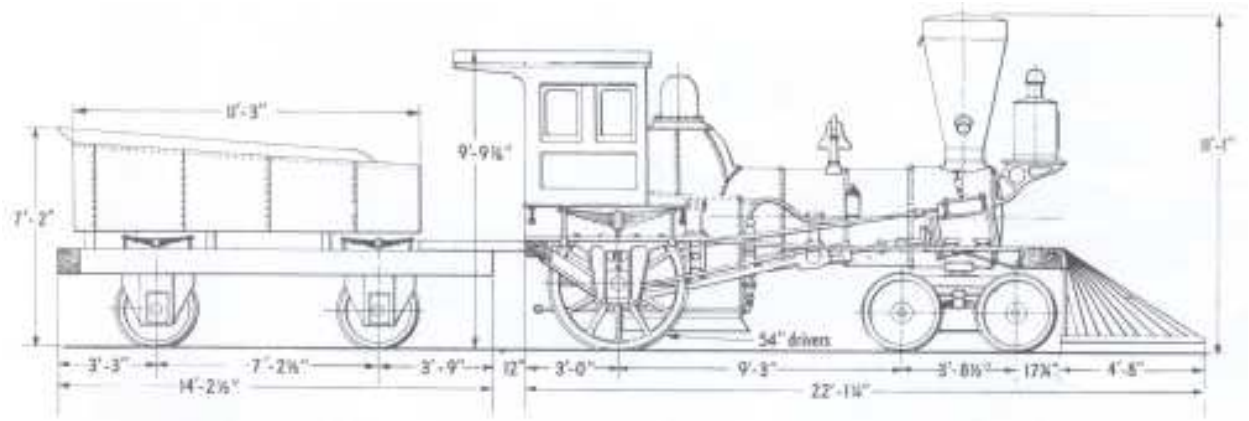
ECE5480 - Modern Computer Architecture

Dr. C. Wang

Project 2

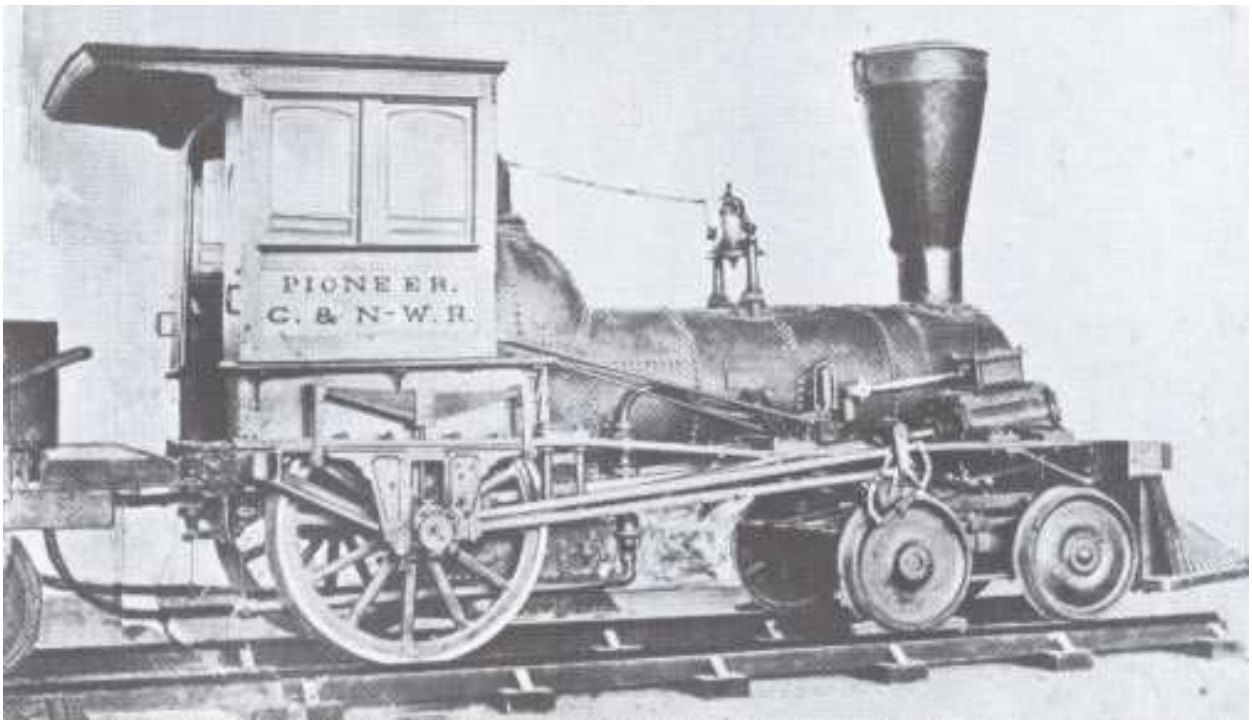
Submitted by Wenton L. Davis

Pioneer



One of the first steam engines built to pull a train was the "Pioneer." It was a very small engine with 4 pilot wheels and 2 drive wheels. Because it was designed as a prototype to test future ideas (that turned into over a century of reailroading), it was named "Pioneer." For similar reasons, I have dubbed this processor design as my own "Pioneer."

-Wenton L. Davis



Images from "Model Railroader Cyclopedia, Volume 1, Steam Locomotives," Kalmbach Publishing Co, 1960

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This document is the design document of the "Pioneer-2" microprocessor. It is designed primarily as a simple RISC processor, although is does implement a few features of some CISC processors. It is an original design, although some of the ideas were features found in other microprocessors, including the Intel®80x86 family and the Analog Devices®ADSP-2100 family.

The structure of this document is to begin with introductory descriptions and definitions and notation descriptions, followed by a simple overview of the processor's main components. A more detailed description of each component is given, then process flow and control functions are described. The instruction set is detailed followed by advanced features (maybe?).

1.1 Notations

Fonts/typsetting

typed/coded text is shown in `courier font`.

addressing

MEM[address] refers to the data stored in memory at location determined by address. For the priliminary form of the Pioneer-2, the memory is split over two sements, the Program Memory, and the Data Memory. For this version, PM[address] will refer to the Program Memory, and DM[address] will refer to the Data Memory.

Bit selection is done with subscripting. For example, if an instruction is being decoded, the CPU may look at bit 22 of the instruction. This is denoted using subscript: instruction$_{22}$. If a group if bits is being refered to, it would appear this way: instruction$_{12-18,20}$ refering to the 7 bits, 12-18 (inclusive) and bit 20.

boolean

Boolean operations are always true/false operations. Each value will always have either a TRUE or a FALSE value, each operation will have either a TRUE or FALSE result. In most contexts of binary numbers, each value is created by a number of bits. If "A" is refering to the contents of one register, and "B" is refering to the contents of a second register, ahd some boolean operation if performed on these two values, the result is a bit-wise comparisson of aligned bits. For example, A AND B would refer to A_0 AND B_0, A_1 AND B_1, A_2 AND B_2, etc.

Symbol	Operation	Example
$\wedge$	AND	A $\wedge$B
$\vee$	OR	C $\vee$D
$\oplus$	XOR	E $\oplus$F
$\sim$or !	NOT	$\sim$(A $\vee$B)

Table 1.1: Boolean Symbols

Often, a plus symbol (+) is used to indicate an "OR" operation, and a dot ($\cdot$) is used to refer to the "AND" operation, but these can be confused with arithmetic operations, which also appear in this document. Therefore, the symbols shown in table 1.1 are used to refer to logical operations:

1.2 Numerical Values

Several numerical values are used throughout this document. By default, numbers are assumed to be decimal numbers. However, other numbering systems may be used at various points. In these cases, standard 'C' formats are used:

`0x303AB76F` denotes a hexadecimal number.

`0o11648329472` denotes an octal number.

`0b00100110111101001001` denotes a binary number.

1.3 Glossary/Definitions

CPU - microprocessor/brain part of computer that performs calculations and controls program and system functions.

Register - a latch that stores data inside the CPU.

General purpose register - any one of the registers that are designed to store data to be used in tha calculations performed by the program. These registers are used equally; all of them have the same functions and capabilities as the others. (Will it be possible for any registers to have extras?)

Register set - this is the collection of all of the general purpose registers.

Special purpose register - any one of several registers that is designed to store data in specific applications such as the Program Counter or Condition Code Register.

ALU - component of CPU which performs addition, subtraction, comparisons, and logical operations such as AND, OR, XOR, NOT, etc.

Shifter unit - component of CPU which performs bit-wise shifting in either direction (up/down refers to left/right). It will perform both logical and arithmetic shifts. (rotates?)

Barrel Shifter - unlike a simple shifet that will shift only a single bit at a time, the barrel shifter can shift multiple bits at a time.

Multiplier unit - component of CPU that performs multiplication. (multiply-and-accumulate instructions?)

Address Generator - The address generator is a collection of registers that function together to perform address generation and perform simple address calculations. These are primarily useful for array structures and buffers. This was inspired by the Address Generators in the Analog Device's ADSP-2100 family. (q.v.)

"X-bus," "Y-bus," and "result-bus" - within the CPU, these three busses are responsible for carrying data (not commands) from the registers to the various calculation units, and the answer from the calculation units back to the registers. The "X-bus" and "Y-bus" carry values to the computational units, usually coming directly out from one of the general purpose registers. The "result-bus" carries the results back to the registers, typically to be loaded into one of the registers.

Program Counter (PC) - This special purpose register is used to indicate the address (in Program Memory) of the next instruction to be executed.

Condition Code Register (CCR) - This special purpose register is used to hold the system operations flags such as negative result flag, carry flag, overflow flag, and interrupt control flag.

1.4 Scope

It is assumed that the audience for this paper has a basic understanding of microprocessor functions and operations. This paper discusses each functional component of the CPU, first giving a functional description of each component, then a more detailed operational description of the component.

Another assumption made in this paper is that once the components of the processor are designed, they will be coded into verilog. Each component will be tested, and then all of the components will be integrated into a single processor, and programmed into a Xilinx Spartan3E Field Programmable Gate Array (FPGA).

The current design of the Pioneer-2 is in a state of flux. The first version was a single-cycle CPU, and not all of the following information will be fully implemented in the preliminary versions. The second version is a multi-cycle processor design. The final verison of the Pioneer-2 will include the additional information.

Additionally, blue text is used to indicate future possible expansions planned for the Pioneer-2.

Chapter 2

The Pioneer-2

2.1 Overview

The Pioneer-2 is designed as a 32-bit RISC processor. It has 32 general purpose registers which can be used for calculations. It can address 32 bit address space (4GB) (capable of more if we use segment registers).

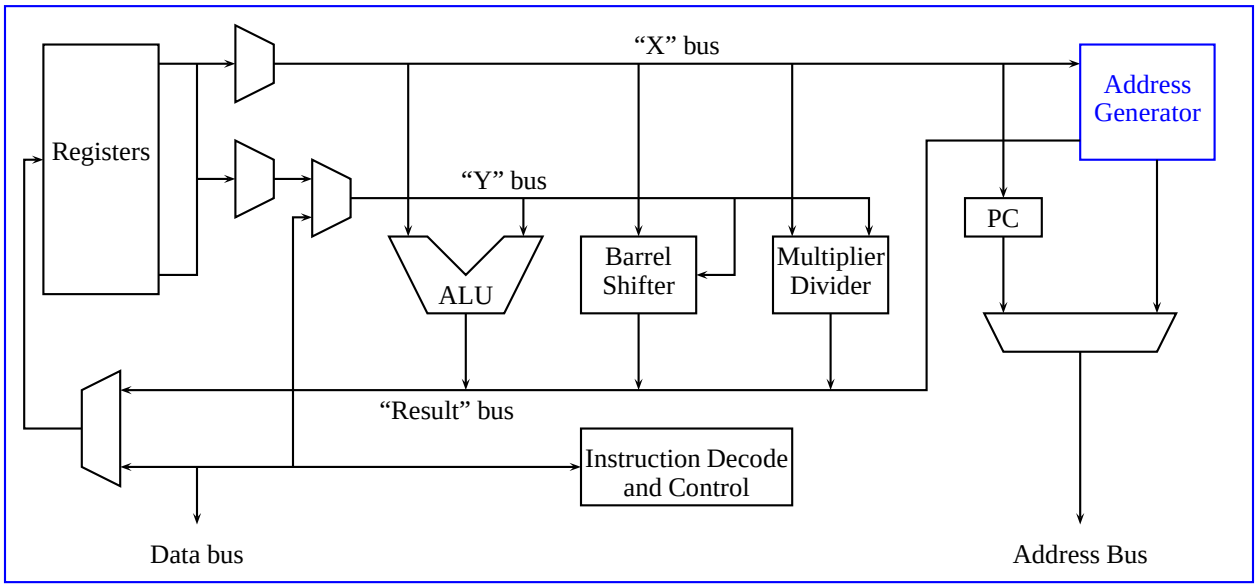


Figure 2.1: Overview

Figure 2.1 shows a simplified block diagram of the Pioneer-2 processor. It shows the internal register set, the "X," "Y," and "result" internal busses, and the computational units. It also shows the Address Generator and instruction decode and control blocks.

Internally, the CPU has a register set, each of which may independantly drive the "X" or "Y" internal data busses. The "Y" bus may alternatively carry immediate data, created by a 16-bit value that may come from the instruction. Once the "X" and "Y" busses have data on them, any of the computational units my access that data and perform calculations on that data. Once the computational unit has accomplished the calculation, the result is placed on the "results" internal data bus, and a register in the register set may load the calculated value.

The Address Generator is used to generate addresses, which can be used to access data within memory MEM[address]. It can be used as an index register. The address generator has the ability to adjust the index registers according to various controls and other registers ("Modify" registers) within the Address Generator. The Address Generator will be able to read the "X" bus to load values into each register in the Address Generator, and the current value of any register (or just A and M registers?) may be placed back onto the "results" bus to be read back into one of the general purpose registers. Generally, however, the output of the Address Generator will be sent to the external address bus. Address Register A7 will be used as the stack pointer.

The Processor Control system is what is responsible for controlling the functions of the CPU. It contains the CCR as well as the program counter (PC) register[1]. It is responsible for sending control signals out to the rest of the processor, as well as many of the I/O signals coming into the processor and going out from the processor. There are a lot of control signals from the control block of the processor to various components in the processor, but for clarity, these signals are not included in ths high-level figure.

[1] In figure 2.1, the PC is shown outside of the control block. This is done for clarity because it's value may be set from the "X" bus and it's output goes through a selector before being placed on the external Address bus.

Chapter 3

Useful Components

There are many components that are used in the design of the Pioneer-2. These components are the basic building blocks of the processor, including adders, decoders, selectors, and simple registers. Various flip-flops and unique circuits are also used throughout the design of the Pioneer-2.

The larger components of the Pioneer-2 are described in seperate chapters, but this chapter is dedicated to describing the smaller components. Each section will describe the smaller components and provide the verilog code that was used to describe the component's modules.

3.1 SR Flip-Flop

The SR flip-flops are very simple storage units, built with a pair of NOR gates. The inputs to the SR flip-flops are two inputs, set, and reset. If the reset input goes high, this forces the q output to a 0, and the qbar output to a 1. If the set input is high, it forces the q output to a 1 and the qbar output to a 0.

One variation on the basic SR flip-flop is the addition of an enable input. This flip-flop will behave the same as the basic SR flip-flop, except that the set and reset signals are masked by the enable input, e. The enable input must be high to allow the set and reset signals to effect the flip-flop.

Verilog code for both types of SR flip-flops are included in the file, `sr_ff_w_gate.v`:

```
module SRlatch ( q, qbar, set, reset );
  output q, qbar;
  input set, reset;

  nor ( q, qbar, reset );
  nor ( qbar, q, set );
endmodule

module SRlatch_e (q, qbar, set, reset, e);
  output q, qbar;
  input set, reset, e;
```

```
  and ( my_reset, reset, e );
  and ( my_set, set, e );

  nor ( q, qbar, my_reset );
  nor ( qbar, q, my_set );
endmodule
```

One small drawback of the SR flip-flops is timing. While the enable input is high, the flip-flop will respond to any changes seen at the set and reset inputs. To provide reliable storage and behavior, the set and reset inputs should be held steady during the time the enable input is high.

The SR flip flop was tested using the following verilog module. This module tests the flip-flop by changing the set and reset signals without the enable being active to prove the enable is required for the flip-flop to respond, then the enable is made active without an active set or reset to show it does not cause a false change in the flip-flop. Finally, the enable is active and the set and reset signals are tested.

```
module test_sr();

  wire q, qnot; //, j, k, clk, preset_not, clear_not;
  reg  s, r, g;

  //sr_ff test( q, qnot, _s, _r, _g );
  //SRlatch test( q, qnot, s, r );
  SRlatch_e test( q, qnot, s, r, g );

  initial
    begin
      #0;
        s=0; r=0; g=0;

      #2;
        s=0; r=1; g=1;

      #2;
        s=0; r=0; g=0;

      $monitor( "set=%1d, reset=%1d, gate=%1d; q=%1d, /q=%1d",
                                       s, r, g, q, qnot );

      #5;

      $display( "\ntry set and reset without enable" );

      #5  s=1;
      #5  s=0;

      #5  r=1;
      #5  r=0;
      #5;
```

```
      $display( "\ntry enable without set or reset" );

      #5  g=0;
      #5  g=1;
      #5;

      $display( "\ntry set and reset with enable" );

      #5  g=1;
      #2  s=1;
      #2  s=0;

      #6;
      #2  r=1;
      #2  r=0;

      #10 $finish;
    end
endmodule
```

The output of this program can be seen in it's output:

```
set=0, reset=0, gate=0; q=0, /q=1

try set and reset without enable
set=1, reset=0, gate=0; q=0, /q=1
set=0, reset=0, gate=0; q=0, /q=1
set=0, reset=1, gate=0; q=0, /q=1
set=0, reset=0, gate=0; q=0, /q=1

try enable without set or reset
set=0, reset=0, gate=1; q=0, /q=1

try set and reset with enable
set=1, reset=0, gate=1; q=1, /q=0
set=0, reset=0, gate=1; q=1, /q=0
set=0, reset=1, gate=1; q=0, /q=1
set=0, reset=0, gate=1; q=0, /q=1
```

3.2 JK Master-Slave Flip-Flop

The second type of flip-flop used in the Pioneer-2 was the JK Master-Slave Flip-Flop. The JK flip-flop is a more complex flip-flop, dependant on the "J" and "K" inputs as well as the current state of the flip-flop. Additionally, this flip-flop has two seperate stages, a "Master" and a "Slave." This allows the flip-flop to sample ("examine") data on one phase of the input clock cycle, and output the current state of the flip-flop on a different stage of the clock cycle. This eliminates a lot of timing issues that existed with the SR flip-flop. The JK flip-flop also has two inputs to perform

the set and reset functions. To better model industry's JK flip-flops, these inputs have been made active-low, meaning that these two inputs are usually high, but a low on the "preset_not" input will cause the flip-flop to enter the "set" state (q output is a 1 and qnot output is a 0), while a low on the "clear_not" will cause the flip-flop to enter the "clear" state (q output is 0 and qnot output is 1).

The verilog code for the JK Master-Slave flip-flop is found in the `jk_ms_ff.v`:

```
module jk_ms_ff( q, qnot, j, k, clk, preset_not, clear_not );
  output q, qnot;
  input  j, k, clk, preset_not, clear_not;

  nand Jin( j1, qnot, j, clk );
  nand Jmf( j2, preset_not, j1, k2 );
  nand Jsin( j3, j2, ~clk );
  nand Jsf( q, preset_not, j3, qnot );

  nand Kin( k1, clk, k, q );
  nand Kmf( k2, j2, k1, clear_not );
  nand Ksin( k3, ~clk, k2 );
  nand Ksf( qnot, q, k3, clear_not );
endmodule
```

The following verilog module was used to test the JK flip-flop. The module begins by testing the J and K inputs without clock pulses, where the outputs should not change. Next, the active low preset and clear inputs are tested (which do not depend on the clock input). Finally, all of the JK inputs are tested with a simulated clock to test the behavior of the JK flip-flop.

```
module test_jk();

  wire q, qnot; //, j, k, clk, preset_not, clear_not;
  reg  j, k, clk, preset_not, clear_not;

  jk_ms_ff test( q, qnot, j, k, clk, preset_not, clear_not );

  initial
    begin
      j=0; k=0; clk=1; preset_not=1; clear_not=1;

      $monitor(
        "/preset=%1d, /clear=%1d, j=%1d, k=%1d, clk=%1d; q=%1d, /q=%1d",
        preset_not, clear_not, j, k, clk, q, /qnot );

      #10 j=1;
      #2  j=0;

      #10 k=1;
      #2  k=0;
```

```
      $display( "neg pulse on preset" );
      #10 preset_not=0; //create negative pulse
      #2  preset_not=1;

      $display( "neg pulse on clear" );
      #10 clear_not=0; //create negative pulse
      #2  clear_not=1;

      $display( "generate clock pulse, should not change Q, /Q" );
      #10 clk=0;
      #2  clk=1;

      $display( "should set Q" );
      #5  j=1;   //set ff
      #5  clk=0;
      #2  clk=1;

      $display( "should toggle FF twice" );
      #5  k=1;   //toggle ff (to clear, then back to set)
      #5  clk=0;
      #2  clk=1;
      #5  clk=0;
      #2  clk=1;

      $display( "should clear Q" );
      #5  j=0;
      #5  clk=0;
      #2  clk=1;

      #10 $finish;
    end
endmodule
```

Thee output of the test program shows the JK Master-Slave flip-flop working as expected:

```
/preset=1, /clear=1, j=0, k=0, clk=1; q=x, /q=x
/preset=1, /clear=1, j=1, k=0, clk=1; q=x, /q=x
/preset=1, /clear=1, j=0, k=0, clk=1; q=x, /q=x
/preset=1, /clear=1, j=0, k=1, clk=1; q=x, /q=x
neg pulse on preset
/preset=1, /clear=1, j=0, k=0, clk=1; q=x, /q=x
/preset=0, /clear=1, j=0, k=0, clk=1; q=1, /q=0
neg pulse on clear
/preset=1, /clear=1, j=0, k=0, clk=1; q=1, /q=0
/preset=1, /clear=0, j=0, k=0, clk=1; q=0, /q=1
generate clock pulse, should not change Q, /Q
/preset=1, /clear=1, j=0, k=0, clk=1; q=0, /q=1
/preset=1, /clear=1, j=0, k=0, clk=0; q=0, /q=1
should set Q
/preset=1, /clear=1, j=0, k=0, clk=1; q=0, /q=1
```

```
/preset=1, /clear=1, j=1, k=0, clk=1; q=0, /q=1
/preset=1, /clear=1, j=1, k=0, clk=0; q=1, /q=0
should toggle FF twice
/preset=1, /clear=1, j=1, k=0, clk=1; q=1, /q=0
/preset=1, /clear=1, j=1, k=1, clk=1; q=1, /q=0
/preset=1, /clear=1, j=1, k=1, clk=0; q=0, /q=1
/preset=1, /clear=1, j=1, k=1, clk=1; q=0, /q=1
/preset=1, /clear=1, j=1, k=1, clk=0; q=1, /q=0
should clear Q
/preset=1, /clear=1, j=1, k=1, clk=1; q=1, /q=0
/preset=1, /clear=1, j=0, k=1, clk=1; q=1, /q=0
/preset=1, /clear=1, j=0, k=1, clk=0; q=0, /q=1
/preset=1, /clear=1, j=0, k=1, clk=1; q=0, /q=1
```

3.3 Decoder

The _decoder32 module is used to select a single signal of 32 possible outputs. The selected output is active low while all others are high. The output signal is selected with a 5-bit "select" input. There is also an active low "_enable" input. This signal must be low for the decoder output to be selected. if the enable input is high, all of the y outputs will be high, regardless of the select input.

The decoder32 module is found in `decoder.v`. This module is currently modeled in behavioral mode:

```
module _decoder32( _y, select, _enable );
  output [31:0] _y;
  input  [4:0]  select;
  input  _enable;

  reg [31:0] _y;

  always @(_enable or select)
    begin
      if( _enable == 0 )
        case( select )
           0  : _y=32'b11111111111111111111111111111110;
           1  : _y=32'b11111111111111111111111111111101;
           2  : _y=32'b11111111111111111111111111111011;
           3  : _y=32'b11111111111111111111111111110111;
           4  : _y=32'b11111111111111111111111111101111;
           5  : _y=32'b11111111111111111111111111011111;
           6  : _y=32'b11111111111111111111111110111111;
           7  : _y=32'b11111111111111111111111101111111;
           8  : _y=32'b11111111111111111111111011111111;
           9  : _y=32'b11111111111111111111110111111111;
          10  : _y=32'b11111111111111111111101111111111;
          11  : _y=32'b11111111111111111111011111111111;
          12  : _y=32'b11111111111111111110111111111111;
          13  : _y=32'b11111111111111111101111111111111;
```

```
          14  : _y=32'b11111111111111111011111111111111;
          15  : _y=32'b11111111111111110111111111111111;
          16  : _y=32'b11111111111111101111111111111111;
          17  : _y=32'b11111111111111011111111111111111;
          18  : _y=32'b11111111111110111111111111111111;
          19  : _y=32'b11111111111101111111111111111111;
          20  : _y=32'b11111111111011111111111111111111;
          21  : _y=32'b11111111110111111111111111111111;
          22  : _y=32'b11111111101111111111111111111111;
          23  : _y=32'b11111111011111111111111111111111;
          24  : _y=32'b11111110111111111111111111111111;
          25  : _y=32'b11111101111111111111111111111111;
          26  : _y=32'b11111011111111111111111111111111;
          27  : _y=32'b11110111111111111111111111111111;
          28  : _y=32'b11101111111111111111111111111111;
          29  : _y=32'b11011111111111111111111111111111;
          30  : _y=32'b10111111111111111111111111111111;
          31  : _y=32'b01111111111111111111111111111111;
          default : _y=32'b11111111111111111111111111111111;
        endcase
      else
        _y=32'hFFFFFFFF;
    end
endmodule
```

The decoder32 module was tested with the following module. The test module executes a loop, counting from 0 to 32. For each count, it clears the enable input to check the selected output being low, then sets the enable signal to make sure the selected output returns high.

```
module test_decoder();
  integer count=0;
  reg     _enable=1;
  wire    [31:0] result;

  _decoder32 test( result, count[4:0], _enable );

  initial
    begin
      $monitor(
        "@ %5d: count=%5X /E=%b ; result=%32b",
        $time, count, _enable, result );

      for( count=0; count < 32; count=count+1 )
        begin
          #1;
          _enable = 0;
          #2;
          _enable = 1;
        end //for(count)
    end //initial
```

```
endmodule
```

The output of the decode32 test module shows the decoder working:

```
@     0: count=00000000 /E=1 ; result=11111111111111111111111111111111
@     1: count=00000000 /E=0 ; result=11111111111111111111111111111110
@     3: count=00000001 /E=1 ; result=11111111111111111111111111111111
@     4: count=00000001 /E=0 ; result=11111111111111111111111111111101
@     6: count=00000002 /E=1 ; result=11111111111111111111111111111111
@     7: count=00000002 /E=0 ; result=11111111111111111111111111111011
@     9: count=00000003 /E=1 ; result=11111111111111111111111111111111
@    10: count=00000003 /E=0 ; result=11111111111111111111111111110111
@    12: count=00000004 /E=1 ; result=11111111111111111111111111111111
@    13: count=00000004 /E=0 ; result=11111111111111111111111111101111
@    15: count=00000005 /E=1 ; result=11111111111111111111111111111111
@    16: count=00000005 /E=0 ; result=11111111111111111111111111011111
@    18: count=00000006 /E=1 ; result=11111111111111111111111111111111
@    19: count=00000006 /E=0 ; result=11111111111111111111111110111111
@    21: count=00000007 /E=1 ; result=11111111111111111111111111111111
@    22: count=00000007 /E=0 ; result=11111111111111111111111101111111
@    24: count=00000008 /E=1 ; result=11111111111111111111111111111111
@    25: count=00000008 /E=0 ; result=11111111111111111111111011111111
@    27: count=00000009 /E=1 ; result=11111111111111111111111111111111
@    28: count=00000009 /E=0 ; result=11111111111111111111110111111111
@    30: count=0000000a /E=1 ; result=11111111111111111111111111111111
@    31: count=0000000a /E=0 ; result=11111111111111111111101111111111
@    33: count=0000000b /E=1 ; result=11111111111111111111111111111111
@    34: count=0000000b /E=0 ; result=11111111111111111111011111111111
@    36: count=0000000c /E=1 ; result=11111111111111111111111111111111
@    37: count=0000000c /E=0 ; result=11111111111111111110111111111111
@    39: count=0000000d /E=1 ; result=11111111111111111111111111111111
@    40: count=0000000d /E=0 ; result=11111111111111111101111111111111
@    42: count=0000000e /E=1 ; result=11111111111111111111111111111111
@    43: count=0000000e /E=0 ; result=11111111111111111011111111111111
@    45: count=0000000f /E=1 ; result=11111111111111111111111111111111
@    46: count=0000000f /E=0 ; result=11111111111111110111111111111111
@    48: count=00000010 /E=1 ; result=11111111111111111111111111111111
@    49: count=00000010 /E=0 ; result=11111111111111101111111111111111
@    51: count=00000011 /E=1 ; result=11111111111111111111111111111111
@    52: count=00000011 /E=0 ; result=11111111111111011111111111111111
@    54: count=00000012 /E=1 ; result=11111111111111111111111111111111
@    55: count=00000012 /E=0 ; result=11111111111110111111111111111111
@    57: count=00000013 /E=1 ; result=11111111111111111111111111111111
@    58: count=00000013 /E=0 ; result=11111111111101111111111111111111
@    60: count=00000014 /E=1 ; result=11111111111111111111111111111111
@    61: count=00000014 /E=0 ; result=11111111111011111111111111111111
@    63: count=00000015 /E=1 ; result=11111111111111111111111111111111
@    64: count=00000015 /E=0 ; result=11111111110111111111111111111111
@    66: count=00000016 /E=1 ; result=11111111111111111111111111111111
```

```
@    67: count=00000016 /E=0 ; result=11111111101111111111111111111111
@    69: count=00000017 /E=1 ; result=11111111111111111111111111111111
@    70: count=00000017 /E=0 ; result=11111111011111111111111111111111
@    72: count=00000018 /E=1 ; result=11111111111111111111111111111111
@    73: count=00000018 /E=0 ; result=11111110111111111111111111111111
@    75: count=00000019 /E=1 ; result=11111111111111111111111111111111
@    76: count=00000019 /E=0 ; result=11111101111111111111111111111111
@    78: count=0000001a /E=1 ; result=11111111111111111111111111111111
@    79: count=0000001a /E=0 ; result=11111011111111111111111111111111
@    81: count=0000001b /E=1 ; result=11111111111111111111111111111111
@    82: count=0000001b /E=0 ; result=11110111111111111111111111111111
@    84: count=0000001c /E=1 ; result=11111111111111111111111111111111
@    85: count=0000001c /E=0 ; result=11101111111111111111111111111111
@    87: count=0000001d /E=1 ; result=11111111111111111111111111111111
@    88: count=0000001d /E=0 ; result=11011111111111111111111111111111
@    90: count=0000001e /E=1 ; result=11111111111111111111111111111111
@    91: count=0000001e /E=0 ; result=10111111111111111111111111111111
@    93: count=0000001f /E=1 ; result=11111111111111111111111111111111
@    94: count=0000001f /E=0 ; result=01111111111111111111111111111111
@    96: count=00000020 /E=1 ; result=11111111111111111111111111111111
```

Eventually, the _decoder32 module will be converted to a structural model instead of a behavioral model.

3.4 Selector

There are three selector modules, “one_of_two32,” “one_of_four32,” and “one_of_eight32.” These three modules all behave similarly to each other. They each have 32-bit inputs. The one_of_two module has two 32-bit inputs, one_of_four has four32-bit inputs, and one_of_eight has eight 32-bit inputs. All of the modules have a single 32-bit output, and a select input that is one, two, or three bits, respectively. The select bits control which of the 32-bit inputs are to be copied to the 32-bit output.

All of the selectors are found in `selector32.v`:

```
module select_one_of_two32( y, a, b, select );
  output [31:0] y;
  input         select;
  input  [31:0] a, b;

  reg [31:0] y;

  always @(select or a or b)
    begin
      y = (select==0)?a:b;
    end
endmodule

module select_one_of_four32( y, a, b, c, d, select );
  output [31:0] y;
```

```
  input   [1:0] select;
  input  [31:0] a, b, c, d;

  reg [31:0] y;

  always @(select or a or b or c or d)
    case ( select )
      0 : y = a;
      1 : y = b;
      2 : y = c;
      3 : y = d;
    endcase
endmodule

module select_one_of_eight32(
  in0, in1, in2, in3, in4, in5, in6, in7, select );
  output [31:0] out;
  input  [31:0] in0, in1, in2, in3, in4, in5, in6, in7;
  input  [2:0]  select;

  reg [31:0] out;

  always @( select or in0 or in1  or in2 or in3 or in4 or in5 or in6 or in7 )
    begin
      case( select )
        0 : out = in0;
        1 : out = in1;
        2 : out = in2;
        3 : out = in3;
        4 : out = in4;
        5 : out = in5;
        6 : out = in6;
        7 : out = in7;
      endcase
    end //always
endmodule
```

The selector modules are all the same, so only the one_of_two32 module was tested with the following verilog module:

```
module test_selector32();
  reg  [31:0] a, b;
  reg  select=0;
  wire [31:0] result;

  select_one_of_two32 test( result, a, b, select );

  initial
    begin
      a=32'hAAAAAAAA;
      b=32'hBBBBBBBB;
```

```
      $monitor( "@ %5d: result=%X select=%b", $time, result, select );

      #5;
      select=1;
      #2;

      $finish;

    end //initial

endmodule
```

The output of the selector test shows the selection between the teo inputs being controlled by the single select bit:

```
@     0: result=aaaaaaaa select=0
@     5: result=bbbbbbbb select=1
```

All of the select modules are built using behavioral modeling instead of structural modeling, but will be converted to structrual modeling eventually.

3.5 Simple Register

A simple register is built using a collection of 32 SR flop-flop based latches. This register is designed specifically as a 32-bit holder of data. These are used throughout the Pioneer-2to hold 32-bit words of data at critial points such as the output of the PC register, and inputs to most of the computational units. The Address Generator will be a collection of these simple registers, but the Pioneer-2's main data registers are more complicated esigns, described in another chapter. The simple reg32 module has a 32-bit input, a 32-bit output, and a single load input. While the input is high, the 32 internal latches are copying the data seen at the input, but when the load signal goes low, the latches hold the current data, regardless of changing input data.

The reg32 module is found in `simplereg.v`:

```
module reg32( out, in, load );
  output [31:0] out;
  input  [31:0] in;
  input  load;

  wire [31:0] nc; //no connections for qbar outputs from each SR FF

  SRlatch_e  bit0 (  out[0],  nc[0],  in[0],  ~in[0], load );
  SRlatch_e  bit1 (  out[1],  nc[1],  in[1],  ~in[1], load );
  SRlatch_e  bit2 (  out[2],  nc[2],  in[2],  ~in[2], load );
```

```
  SRlatch_e  bit3 (  out[3],  nc[3],  in[3],  ~in[3], load );
  SRlatch_e  bit4 (  out[4],  nc[4],  in[4],  ~in[4], load );
  SRlatch_e  bit5 (  out[5],  nc[5],  in[5],  ~in[5], load );
  SRlatch_e  bit6 (  out[6],  nc[6],  in[6],  ~in[6], load );
  SRlatch_e  bit7 (  out[7],  nc[7],  in[7],  ~in[7], load );
  SRlatch_e  bit8 (  out[8],  nc[8],  in[8],  ~in[8], load );
  SRlatch_e  bit9 (  out[9],  nc[9],  in[9],  ~in[9], load );
  SRlatch_e bit10 ( out[10], nc[10], in[10], ~in[10], load );
  SRlatch_e bit11 ( out[11], nc[11], in[11], ~in[11], load );
  SRlatch_e bit12 ( out[12], nc[12], in[12], ~in[12], load );
  SRlatch_e bit13 ( out[13], nc[13], in[13], ~in[13], load );
  SRlatch_e bit14 ( out[14], nc[14], in[14], ~in[14], load );
  SRlatch_e bit15 ( out[15], nc[15], in[15], ~in[15], load );
  SRlatch_e bit16 ( out[16], nc[16], in[16], ~in[16], load );
  SRlatch_e bit17 ( out[17], nc[17], in[17], ~in[17], load );
  SRlatch_e bit18 ( out[18], nc[18], in[18], ~in[18], load );
  SRlatch_e bit19 ( out[19], nc[19], in[19], ~in[19], load );
  SRlatch_e bit20 ( out[20], nc[20], in[20], ~in[20], load );
  SRlatch_e bit21 ( out[21], nc[21], in[21], ~in[21], load );
  SRlatch_e bit22 ( out[22], nc[22], in[22], ~in[22], load );
  SRlatch_e bit23 ( out[23], nc[23], in[23], ~in[23], load );
  SRlatch_e bit24 ( out[24], nc[24], in[24], ~in[24], load );
  SRlatch_e bit25 ( out[25], nc[25], in[25], ~in[25], load );
  SRlatch_e bit26 ( out[26], nc[26], in[26], ~in[26], load );
  SRlatch_e bit27 ( out[27], nc[27], in[27], ~in[27], load );
  SRlatch_e bit28 ( out[28], nc[28], in[28], ~in[28], load );
  SRlatch_e bit29 ( out[29], nc[29], in[29], ~in[29], load );
  SRlatch_e bit30 ( out[30], nc[30], in[30], ~in[30], load );
  SRlatch_e bit31 ( out[31], nc[31], in[31], ~in[31], load );
endmodule
```

Testing the reg32 module shows the load signal begin held high, and the output data simply follows the input values, but once the load signal is low, the output values do not change, even when the input data does.

```
module test_simple();

  reg [31:0] feed=32'h01234567;
  reg enable=1'b0;
  wire [31:0] read;

  reg32 test ( read, feed, enable );

  initial
    begin
      #0;
        $monitor( "%5d: [%8x]->(%b)->[%8x]", $time, feed, enable, read );
      #2;
        feed=32'h98765432;
      #3;
        enable=1;
```

```
      #1;
        enable=0;
      #10;
        feed=32'hfedcba98;
      #2;
        enable = 1;
      #5;
        feed=32'h1c2d3e5f;
      #10;
        enable = 0;
      #10;
        $finish;
    end //initial
endmodule
```

The output of the test shows the data being held:

```
    0: [01234567]->(0)->[xxxxxxxx]
    2: [98765432]->(0)->[xxxxxxxx]
    5: [98765432]->(1)->[98765032]
    6: [98765432]->(0)->[98765032]
   16: [fedcba98]->(0)->[98765032]
   18: [fedcba98]->(1)->[fedcba98]
   23: [1c2d3e5f]->(1)->[1c2d3e5f]
   33: [1c2d3e5f]->(0)->[1c2d3e5f]
```

3.6 Static RAM

The Pioneer-2 has two memory segments, data memory and program memory. Although the program memory is not written to, the data memory is written. For simplicity, both memory segments are using the "sram" module[1]. The sram module has a 32-bit input, "d_in" for 32 bits of input data, a 10-bit input, "address" to determine which 32-bit word to select[2],and an active high input signal, "write" used to write data into memory, and "select" input which must be high to enable the sram module. The sram module has a 32-bit data output. If select is not high, the 32-bit output will be high-impedance.

The sram module is found tin the file, `sram.v`[3]:

```
module sram( d_out, d_in, address, write, select );
  output [31:0] d_out;
```

[1] In future versions, both memory segments will be controlled through a cache memory system.

[2] The initial version used only 1K word memory segments, but this could easily be increased if necessary.

[3] The $display() system call should not be necessary, but while debuging the system, it caused the module to work correctly. Commenting it back out caused the module to begin to fail. A bug has been filed with the software developeers, but no response has been received as of yet. The test module worked fine without the $display() call, so the blank lines do not appear in the test output. When the module is converted to a structural model instead of behaioral, this issue will not exist.

```
  input  [31:0] d_in;
  input  [9:0]  address;
  input         write;
  input         select;

  reg [31:0] mem [1023:0];//memory
  reg [31:0] d_out;

  always @( d_in, address, write, select )
    begin
      if( write && select )
        begin
          mem[ address ] = d_in;
          $display( "" );
        end

      assign d_out = select ? mem[ address ] : 32'hzzzzzzzz;
    end

endmodule
```

The sram module was tested with the following module. It executes a loop which fills the memory segment with the address value being stored in the corresponding 32-bit word in memory (0-extended for the high bits). It writes the 1K-word memory values out to a file named "test.ram." Then, is randomly selects 20 addresses and prints their contents[4].

```
module test_sram();

  wire[31:0] data;
  reg [31:0] x_bus;
  reg [9:0]  addr;
  reg        write=1'b0, select=1'b0;
  integer    i;

  sram testram ( data, x_bus, addr, write, select );

  initial
    #0 begin
    addr = 10'h000;
    select = 1'b1;
    $monitor( "Addr:%3X data:%8X WR%b", addr, data, write );

    for( i=0; i<1024; i=i+1 )
      begin
        x_bus = 32'h00000000 | i;
        addr  = i;
        #1 write = 1'b1;
        #1 write = 0;
```

[4]This accomplished two things: it provided a file with the exact format needed by verilog to fill the program memory later in development, and it also provided a simple experiment with the $random system call.

```
      end

    $writememh( "test.mem", testram.mem );
    #1;

    for( i=0; i<20; i=i+1 )
      begin
        addr = $random;
        #10;
      end

    //make sure data is not written when select is low...
    select = 0;
    addr=0;
    x_bus=32'hFFFF0000;
    #1;
    write=1;
    #1;
    write=0;
    select=1;

    #1
    $finish;
  end

endmodule
```

The output of the sram test has most of it's output cut out. It shows the first few writes into the memory, then the last few writes, followed by the 20 random outputs. Finally, it shows the high-impedance outputs when the memory is not selected, regardless of the state of the write signal.

```
Addr:000 data:xxxxxxxx WR0
Addr:000 data:00000000 WR1
Addr:001 data:xxxxxxxx WR0
Addr:001 data:00000001 WR1
Addr:002 data:xxxxxxxx WR0
Addr:002 data:00000002 WR1
Addr:003 data:xxxxxxxx WR0
Addr:003 data:00000003 WR1
Addr:004 data:xxxxxxxx WR0
Addr:004 data:00000004 WR1

(...)

Addr:3fd data:xxxxxxxx WR0
Addr:3fd data:000003fd WR1
Addr:3fe data:xxxxxxxx WR0
Addr:3fe data:000003fe WR1
Addr:3ff data:xxxxxxxx WR0
Addr:3ff data:000003ff WR1
```

```
Addr:3ff data:000003ff WR0
Addr:124 data:00000124 WR0
Addr:281 data:00000281 WR0
Addr:209 data:00000209 WR0
Addr:263 data:00000263 WR0
Addr:30d data:0000030d WR0
Addr:18d data:0000018d WR0
Addr:065 data:00000065 WR0
Addr:212 data:00000212 WR0
Addr:301 data:00000301 WR0
Addr:10d data:0000010d WR0
Addr:176 data:00000176 WR0
Addr:13d data:0000013d WR0
Addr:3ed data:000003ed WR0
Addr:38c data:0000038c WR0
Addr:1f9 data:000001f9 WR0
Addr:0c6 data:000000c6 WR0
Addr:0c5 data:000000c5 WR0
Addr:2aa data:000002aa WR0
Addr:3e5 data:000003e5 WR0
Addr:277 data:00000277 WR0
Addr:000 data:zzzzzzzz WR0
Addr:000 data:zzzzzzzz WR1
Addr:000 data:00000000 WR0
```

The sram module is built as a behavioral modal, but will eventually be converted to a structural model.

3.7 Adders

The fast_adder module uses a carry look-ahead generator, based on Texas Intrument's documentation of the 7483 adder. The adder has two 32-bit inputs, "a" and "b" that provide the two numbers to add, as well as a single bit "c_in" input carry bit. It provides a 32-bit output result, "sum," and a single bit carry out, "c_out."

The verilog code for the adder is found in the file, `adder32.v`, but because of the size of this file, only a smaller, 7-bit adder used in prototyping is shown in this listing:

```
module fast_adder( sum, c_out, a, b, c_in );
  output [7:0] sum;
  output       c_out;
  input  [7:0] a;
  input  [7:0] b;
  input  c_in;

  wire [7:0] o; //output of NOR gates
  wire [7:0] n; //output of NAND gates
  wire [7:0] bitsum; //and n[x] with !o[x] feed upper part of XOR
  wire [7:0] bitinv; //lower half feed into XOR
```

```
    wire cnot;    //inverted c_in

    //n[7:0] = a[7:0] ~& b[7:0]; //NAND each input pair
    nand (n[0],a[0],b[0]);
    nand (n[1],a[1],b[1]);
    nand (n[2],a[2],b[2]);
    nand (n[3],a[3],b[3]);
    nand (n[4],a[4],b[4]);
    nand (n[5],a[5],b[5]);
    nand (n[6],a[6],b[6]);
    nand (n[7],a[7],b[7]);

    //o[7:0] = a[7:0] ~| b[7:0]; //NOR each input pair
    nor (o[0],a[0],b[0]);
    nor (o[1],a[1],b[1]);
    nor (o[2],a[2],b[2]);
    nor (o[3],a[3],b[3]);
    nor (o[4],a[4],b[4]);
    nor (o[5],a[5],b[5]);
    nor (o[6],a[6],b[6]);
    nor (o[7],a[7],b[7]);

    not (cnot,c_in);

    //build the output bits
    //bitsum[7:0] = n[7:0] & ~o[7:0];
    and (bitsum[0],n[0],~o[0]);
    and (bitsum[1],n[1],~o[1]);
    and (bitsum[2],n[2],~o[2]);
    and (bitsum[3],n[3],~o[3]);
    and (bitsum[4],n[4],~o[4]);
    and (bitsum[5],n[5],~o[5]);
    and (bitsum[6],n[6],~o[6]);
    and (bitsum[7],n[7],~o[7]);

    not (bitinv[0],cnot);

    and (a_0_0,cnot,n[0]);
    nor (bitinv[1],o[0],a_0_0);

    and (a_1_0,cnot,n[0],n[1]);
    and (a_1_1,o[0],n[1]);
    nor (bitinv[2],o[1],a_1_0,a_1_1);

    and (a_2_0,cnot,n[0],n[1],n[2]);
    and (a_2_1,o[0],n[1],n[2]);
    and (a_2_2,o[1],n[2]);
    nor (bitinv[3],o[2],a_2_0,a_2_1,a_2_2);

    and (a_3_0,cnot,n[0],n[1],n[2],n[3]);
    and (a_3_1,o[0],n[1],n[2],n[3]);
    and (a_3_2,o[1],n[2],n[3]);
    and (a_3_3,o[2],n[3]);
    nor (bitinv[4],o[3],a_3_0,a_3_1,a_3_2,a_3_3);
```

```
  and (a_4_0,cnot,n[0],n[1],n[2],n[3],n[4]);
  and (a_4_1,o[0],n[1],n[2],n[3],n[4]);
  and (a_4_2,o[1],n[2],n[3],n[4]);
  and (a_4_3,o[2],n[3],n[4]);
  and (a_4_4,o[3],n[4]);
  nor (bitinv[5],o[4],a_4_0,a_4_1,a_4_2,a_4_3,a_4_4);

  and (a_5_0,cnot,n[0],n[1],n[2],n[3],n[4],n[5]);
  and (a_5_1,o[0],n[1],n[2],n[3],n[4],n[5]);
  and (a_5_2,o[1],n[2],n[3],n[4],n[5]);
  and (a_5_3,o[2],n[3],n[4],n[5]);
  and (a_5_4,o[3],n[4],n[5]);
  and (a_5_5,o[4],n[5]);
  nor (bitinv[6],o[5],a_5_0,a_5_1,a_5_2,a_5_3,a_5_4,a_5_5);

  and (a_6_0,cnot,n[0],n[1],n[2],n[3],n[4],n[5],n[6]);
  and (a_6_1,o[0],n[1],n[2],n[3],n[4],n[5],n[6]);
  and (a_6_2,o[1],n[2],n[3],n[4],n[5],n[6]);
  and (a_6_3,o[2],n[3],n[4],n[5],n[6]);
  and (a_6_4,o[3],n[4],n[5],n[6]);
  and (a_6_5,o[4],n[5],n[6]);
  and (a_6_6,o[5],n[6]);
  nor (bitinv[7],o[6],a_6_0,a_6_1,a_6_2,a_6_3,a_6_4,a_6_5,a_6_6);

  and (a_7_0,cnot,n[0],n[1],n[2],n[3],n[4],n[5],n[6],n[7]);
  and (a_7_1,o[0],n[1],n[2],n[3],n[4],n[5],n[6],n[7]);
  and (a_7_2,o[1],n[2],n[3],n[4],n[5],n[6],n[7]);
  and (a_7_3,o[2],n[3],n[4],n[5],n[6],n[7]);
  and (a_7_4,o[3],n[4],n[5],n[6],n[7]);
  and (a_7_5,o[4],n[5],n[6],n[7]);
  and (a_7_6,o[5],n[6],n[7]);
  and (a_7_7,o[6],n[7]);
  nor (c_out,    o[7],a_7_0,a_7_1,a_7_2,a_7_3,a_7_4,a_7_5,a_7_6,a_7_7);

  xor (sum[0],bitinv[0],bitsum[0]);
  xor (sum[1],bitinv[1],bitsum[1]);
  xor (sum[2],bitinv[2],bitsum[2]);
  xor (sum[3],bitinv[3],bitsum[3]);
  xor (sum[4],bitinv[4],bitsum[4]);
  xor (sum[5],bitinv[5],bitsum[5]);
  xor (sum[6],bitinv[6],bitsum[6]);
  xor (sum[7],bitinv[7],bitsum[7]);
endmodule
```

The adder was tested using the following verilog module:

```
module test_adder();

  reg [7:0] a,b;
```

```
  reg carry_feed;

  wire carry_out;
  wire [7:0] sum;

  fast_adder tested( sum, carry_out, a, b, carry_feed );

  initial
    begin
      a=0; b=0; carry_feed=0;

      $monitor( $time, "  a=%2X, + b=%2X, cin=%1d : sum=%2X, cout=%1d",
                a, b, carry_feed, sum, carry_out );

      #100 carry_feed=1; //set carry
      #10  carry_feed=0; //clear carry

      #40  a=8'h5A; b=8'h3C;
      #5   carry_feed=1; //set carry
      #5   carry_feed=0; //clear carry
      #40  a=0;b=0;

      #40  a=8'hAC; b=8'h6F;
      #5   carry_feed=1; //set carry
      #5   carry_feed=0; //clear carry
      #40  a=0;b=0;

      #10 $finish;
    end
endmodule
```

The output from the fast_add test modules produces the output:

```
                  0  a=00, + b=00, cin=0 : sum=00, cout=0
                100  a=00, + b=00, cin=1 : sum=01, cout=0
                110  a=00, + b=00, cin=0 : sum=00, cout=0
                150  a=5a, + b=3c, cin=0 : sum=96, cout=0
                155  a=5a, + b=3c, cin=1 : sum=97, cout=0
                160  a=5a, + b=3c, cin=0 : sum=96, cout=0
                200  a=00, + b=00, cin=0 : sum=00, cout=0
                240  a=ac, + b=6f, cin=0 : sum=1b, cout=1
                245  a=ac, + b=6f, cin=1 : sum=1c, cout=1
                250  a=ac, + b=6f, cin=0 : sum=1b, cout=1
                290  a=00, + b=00, cin=0 : sum=00, cout=0
```

The 32-bit version of the adder was very similar to the 8-bit version, and provided similar test results.

3.8 Adder/Subtractor

The final individual component un the Pioneer-2 was the adder/subtractor. More than just the adder, the adder/subtractor provided the ability to perform subtraction as well as addition. It performs this task by adding ad extra input signal, “sub_add” which will determine if addition or subtraction is to be performed. If this extra input is a 0, addition will be performed. If this input is a 1, subtraction is performed. It performs the subtraction by creating a form of two’s compliment. It inverts all of the bits of the “b” input, forming a one’s compliment, and it inverts the carry input signal to add 1, completing the two’s compliment[5]. The modified inputs are then put into a 32-bit fast adder.

The adder/subtractor module is found in the file, addsub32.v:

```
module fast_addsub32( sum, c_out, a, b_in, c_in, sub_add );
  output [31:0] sum;
  output       c_out;
  input  [31:0] a;
  input  [31:0] b_in;
  input  c_in;
  input  sub_add; //1=subtract(a-b),0=add

  wire [31:0] o; //output of NOR gates
  wire [31:0] n; //output of NAND gates
  wire [31:0] bitsum; //and n[x] with !o[x] feed upper part of XOR
  wire [31:0] bitinv; //lower half feed into XOR
  wire cnot;    //inverted c_in
  wire [31:0] b;
  wire cinxorsub;

  xor (cinxorsub, c_in, sub_add); //invert c_in for subtraction

  xor ( b[0], b_in[0],sub_add);
  xor ( b[1], b_in[1],sub_add);
  xor ( b[2], b_in[2],sub_add);
  xor ( b[3], b_in[3],sub_add);
  xor ( b[4], b_in[4],sub_add);
  xor ( b[5], b_in[5],sub_add);
  xor ( b[6], b_in[6],sub_add);
  xor ( b[7], b_in[7],sub_add);
  xor ( b[8], b_in[8],sub_add);
  xor ( b[9], b_in[9],sub_add);
  xor (b[10],b_in[10],sub_add);
  xor (b[11],b_in[11],sub_add);
  xor (b[12],b_in[12],sub_add);
  xor (b[13],b_in[13],sub_add);
  xor (b[14],b_in[14],sub_add);
  xor (b[15],b_in[15],sub_add);
  xor (b[16],b_in[16],sub_add);
  xor (b[17],b_in[17],sub_add);
  xor (b[18],b_in[18],sub_add);
```

[5] If the carry input were already set, indicating the subtraction (borrow) of a 1, this would cancel the addition of on in the conversion of the two’s compliment.

```
   xor (b[19],b_in[19],sub_add);
   xor (b[20],b_in[20],sub_add);
   xor (b[21],b_in[21],sub_add);
   xor (b[22],b_in[22],sub_add);
   xor (b[23],b_in[23],sub_add);
   xor (b[24],b_in[24],sub_add);
   xor (b[25],b_in[25],sub_add);
   xor (b[26],b_in[26],sub_add);
   xor (b[27],b_in[27],sub_add);
   xor (b[28],b_in[28],sub_add);
   xor (b[29],b_in[29],sub_add);
   xor (b[30],b_in[30],sub_add);
   xor (b[31],b_in[31],sub_add);

(32-bit adder ommited)

endmodule
```

Because the fast_addsub32() module became the heart of the ALU, the testing for this module was covered in that module.

Chapter 4

The Register Set

There are several registers in the Pioneer-2. The general purpose registers are named "R0-R31[1]." Each of these 32-bit registers are used for calculations. All of the registers are treated the same; there are no special-purpose registers in this register set[2]. The complete set of these general purpose registers is refered to as the "register file." The Address Generator has several registers contain in it, which will be discussed in the chapter covering the Address Generator. There are also several special purpose registers associated with different computational units, which will be covered as parts of their appropriate computation units.

There are two data busses in the Pioneer-2 processor that the registers can be driven by any of the general purpose registers. The "X" bus and the "Y" buses are used to carry data from the general purpose registers to be used as inputs (parameters) to the various computational units.

The instruction set allows moving data into any register with an assignment that resembles a simple arithmetic statement:

```
R23 = 0x990
```

This simple statement would move the immediate value, 0x990, into register R23. In the statement above, the left of the = sign is the destination register, where the result is to be stored. When using immediate data, the cpu is limited to 16 bits of data, and values are assumed to be the lowest order 16 bits. Immediate values are sign-extended if the instruction_{22} bit is set, otherwise, the high 16 bits are cleared.

There is a special form of immediate mode addressing mode, however, which will allow the immediate data to be moved into the high-order 16 bits of the destination register. By appending an 'h' (or 'H') to the end of the register name, this indicates that the value being loaded will be placed in the high-order bits:

```
R23h=0x8000
```

Loading a value into the high-order bits will clear the contents of the lower-order bits, so if the two previous intructions were executed (in the order presented), register R23 would have the value 0x80000000.

[1] There may be more registers in the final design, depending on space.

[2] Some processors use various registers for special purpose such as "link" (return address) register, "stack pointer," or "frame pointer," but the Pioneer-2 processor does not use any general purpose registers this way.

4.1 General Purpose Registers

The register set is build with a collection of 32-bit registers. Each register is independant of the other registers, and registers are not combined to form 64-bit words, although software could perform 64-bit operations if necessary.

The registers each have several in-place calculations they may perform. Each register may be cleared, incremented, decremented, or complimented in place. This simplifies that use of the ALU and the internal data busses, and improves the pipelined performance of the Pioneer-2. The following examples show the instruction set commands used to perform these:

```
CLEAR R4  ;Clear register 4
INC R18   ;Increment register 18
DEC R14   ;Decrement register 14
NOT R22   ;One's compliment of register 22
```

The general purpose registers are detailed in Figure 4.1, showing the logical connections allowing the register's functions. Note that only one output, `Q[0]` is shown, but each of the JK flipflops' "Q" output is actually output from the register[3].

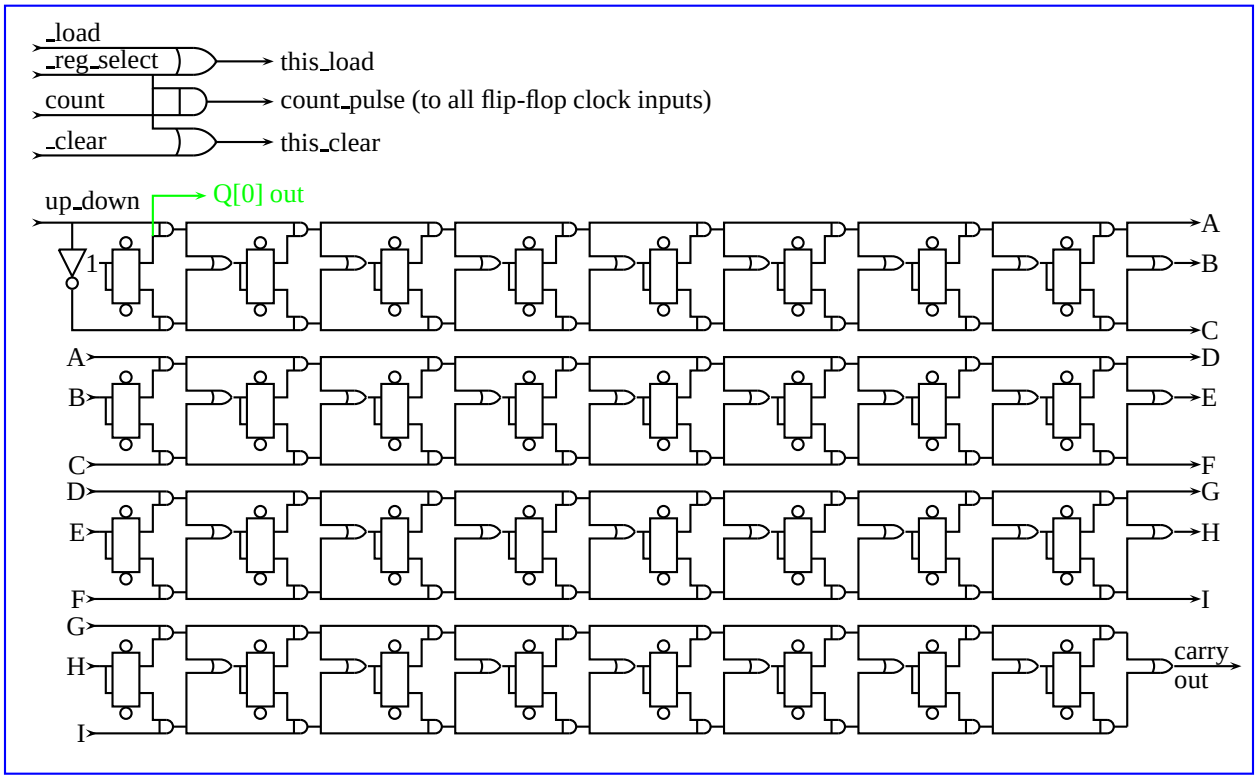


Figure 4.1: General Purpose Register

[3]The register's load circuitry has also been hidden. This circuitry can be found in the TTL specifications book from Texas Instruments, based on the 74191 family of counters.

4.2 Special Purpose Registers

There are also several registers that are not used as general purpose registers. These special purpose registers do not have the same capabilities as the general purpose registers. These special purpose registers are used to store data, which is used in various applications in the Pioneer-2. Table 4.1 shows the special purpose registers.

Number	Register	Number	Register
0	A0	16	L0
1	A1	17	L1
2	A2	18	L2
3	A3	19	L3
4	A4	20	L4
5	A5	21	L5
6	A6	22	L6
7	A7 (Stack Pointer)	23	L7
8	M0	24	MUL_HIGH
9	M1	25	MUL_LOW
10	M2	26	Program Counter (PC)
11	M3	27	Condition Code Reg (CCR)
12	M4	28	BIT_REV
13	M5	29	???
14	M6	30	???
15	M7	31	RANDOM

Table 4.1: Special Purpose Registers

Most of the special purpose registers (A0-A7, M0-M7, L0-L7) are defined in the Address Generator. The MUL_HIGH and MUL_LOW registers are part of the Multiplier/Divider Unit, storing the high 32 bit result of a multiplication, or the dividend or a division in MUL_HIGH, or the low 32 bit result of a multiplication or the remainder of a division in MUL_LOW. BIT_REV is a register that is used to reverse the order of a 32-bit value. The Program Counter, PC, is an index into Program Memory to indicate the address of the next instruction to be executed. All of the system flags are stored in the Condition Code Register, CCR. The RANDOM register is a register that is modified with the system clock, providing randomized data to be read from it.

4.3 Register Code

The verilog code for the registers is in the file, `register.v`. The register module has lots of inputs and outputs. Each register has a 32-bit input to load data, and a 32-bit output. The register also has an active low signal, "_reg_select" that is used to enable the register's functions. If this signal is high, the register will ignore any other input signals, and all of the flag output signals will be disabled. The 32-bit output from the register will still be active, regardless of the _reg_select output. The "_clear" active low input signal will force the register's contents to 0, while the active low "_load" will load the 32-bit input data into the register. The "count" signal provides an active-high clock signal into the register. If the "up_down" signal is high on the rising edge of the count, the register will increment it's value, or it will decrement if the up_down signal is low on the rising edge of the count pulse. However, if the "ones_compliment" signal is high on the rising edge of the count pulse, the up_down signal is ignored, and the register inverts each bit in the register, performing a "NOT," or "one's compliment." In addition to the 32-bit output, there is also an output for the carry flag, negative flag, overflow flag, and zero flag. Each of these outputs is only valid when the register increments, decrements, or inverts.

```
module register( q, carry, negative, overflow, zero,
                 d, _clear, _load, up_down, count, ones_comp, _reg_select );
  output [31:0] q;
  output carry, negative, overflow, zero;
  input  [31:0] d;
  input  _clear, _load, up_down, count, _reg_select, ones_comp;

  wire [31:0] qnot; //each of the Q not outputs
  wire [31:0] _pre; //each of the PRE not inputs
  wire [31:0] _clr; //each of the CLR not inputs
  wire [31:0] up;   //value to be anded with Q output from this stage
  wire [31:0] down; //value anded  with Qnot output from this stage
  wire [31:1] or_from; //output from or from previous bit feed J and K
  wire [31:0] this_d;  //intermediate storage for each d input bit
  wire this_load, count_pulse, this_clear; //global pulses to each bit

  //Generate masked signals to perform actions only when this register is
  //selected
  or (this_load, _load, _reg_select);     //load this register
  and (count_pulse, count, ~_reg_select ); //inc/dec this register
  or (this_clear, _clear, _reg_select );  //clear this register

  //Flip-flops for each bit
  jk_ms_ff  bit0( q[0], qnot[0], 1'b1, 1'b1,
                  count_pulse, _pre[0], _clr[0] );
  jk_ms_ff  bit1( q[1], qnot[1], or_from[1], or_from[1],
                  count_pulse, _pre[1], _clr[1] );
  jk_ms_ff  bit2( q[2], qnot[2], or_from[2], or_from[2],
                  count_pulse, _pre[2], _clr[2] );
  jk_ms_ff  bit3( q[3], qnot[3], or_from[3], or_from[3],
                  count_pulse, _pre[3], _clr[3] );
  jk_ms_ff  bit4( q[4], qnot[4], or_from[4], or_from[4],
                  count_pulse, _pre[4], _clr[4] );
  jk_ms_ff  bit5( q[5], qnot[5], or_from[5], or_from[5],
                  count_pulse, _pre[5], _clr[5] );
  jk_ms_ff  bit6( q[6], qnot[6], or_from[6], or_from[6],
                  count_pulse, _pre[6], _clr[6] );
  jk_ms_ff  bit7( q[7], qnot[7], or_from[7], or_from[7],
                  count_pulse, _pre[7], _clr[7] );
  jk_ms_ff  bit8( q[8], qnot[8], or_from[8], or_from[8],
                  count_pulse, _pre[8], _clr[8] );
  jk_ms_ff  bit9( q[9], qnot[9], or_from[9], or_from[9],
                  count_pulse, _pre[9], _clr[9] );
  jk_ms_ff bit10( q[10], qnot[10], or_from[10], or_from[10],
                  count_pulse, _pre[10], _clr[10] );
  jk_ms_ff bit11( q[11], qnot[11], or_from[11], or_from[11],
                  count_pulse, _pre[11], _clr[11] );
  jk_ms_ff bit12( q[12], qnot[12], or_from[12], or_from[12],
                  count_pulse, _pre[12], _clr[12] );
  jk_ms_ff bit13( q[13], qnot[13], or_from[13], or_from[13],
                  count_pulse, _pre[13], _clr[13] );
  jk_ms_ff bit14( q[14], qnot[14], or_from[14], or_from[14],
```

```
                count_pulse, _pre[14], _clr[14] );
jk_ms_ff bit15( q[15], qnot[15], or_from[15], or_from[15],
                count_pulse, _pre[15], _clr[15] );
jk_ms_ff bit16( q[16], qnot[16], or_from[16], or_from[16],
                count_pulse, _pre[16], _clr[16] );
jk_ms_ff bit17( q[17], qnot[17], or_from[17], or_from[17],
                count_pulse, _pre[17], _clr[17] );
jk_ms_ff bit18( q[18], qnot[18], or_from[18], or_from[18],
                count_pulse, _pre[18], _clr[18] );
jk_ms_ff bit19( q[19], qnot[19], or_from[19], or_from[19],
                count_pulse, _pre[19], _clr[19] );
jk_ms_ff bit20( q[20], qnot[20], or_from[20], or_from[20],
                count_pulse, _pre[20], _clr[20] );
jk_ms_ff bit21( q[21], qnot[21], or_from[21], or_from[21],
                count_pulse, _pre[21], _clr[21] );
jk_ms_ff bit22( q[22], qnot[22], or_from[22], or_from[22],
                count_pulse, _pre[22], _clr[22] );
jk_ms_ff bit23( q[23], qnot[23], or_from[23], or_from[23],
                count_pulse, _pre[23], _clr[23] );
jk_ms_ff bit24( q[24], qnot[24], or_from[24], or_from[24],
                count_pulse, _pre[24], _clr[24] );
jk_ms_ff bit25( q[25], qnot[25], or_from[25], or_from[25],
                count_pulse, _pre[25], _clr[25] );
jk_ms_ff bit26( q[26], qnot[26], or_from[26], or_from[26],
                count_pulse, _pre[26], _clr[26] );
jk_ms_ff bit27( q[27], qnot[27], or_from[27], or_from[27],
                count_pulse, _pre[27], _clr[27] );
jk_ms_ff bit28( q[28], qnot[28], or_from[28], or_from[28],
                count_pulse, _pre[28], _clr[28] );
jk_ms_ff bit29( q[29], qnot[29], or_from[29], or_from[29],
                count_pulse, _pre[29], _clr[29] );
jk_ms_ff bit30( q[30], qnot[30], or_from[30], or_from[30],
                count_pulse, _pre[30], _clr[30] );
jk_ms_ff bit31( q[31], qnot[31], or_from[31], or_from[31],
                count_pulse, _pre[31], _clr[31] );

//row of and gates used in the UP (INC) instruction
and ( up[1], up_down, q[0] );
and ( up[2],  up[1],  q[1] );
and ( up[3],  up[2],  q[2] );
and ( up[4],  up[3],  q[3] );
and ( up[5],  up[4],  q[4] );
and ( up[6],  up[5],  q[5] );
and ( up[7],  up[6],  q[6] );
and ( up[8],  up[7],  q[7] );
and ( up[9],  up[8],  q[8] );
and ( up[10], up[9],  q[9] );
and ( up[11], up[10], q[10] );
and ( up[12], up[11], q[11] );
and ( up[13], up[12], q[12] );
and ( up[14], up[13], q[13] );
and ( up[15], up[14], q[14] );
and ( up[16], up[15], q[15] );
and ( up[17], up[16], q[16] );
```

```
    and ( up[18], up[17], q[17] );
    and ( up[19], up[18], q[18] );
    and ( up[20], up[19], q[19] );
    and ( up[21], up[20], q[20] );
    and ( up[22], up[21], q[21] );
    and ( up[23], up[22], q[22] );
    and ( up[24], up[23], q[23] );
    and ( up[25], up[24], q[24] );
    and ( up[26], up[25], q[25] );
    and ( up[27], up[26], q[26] );
    and ( up[28], up[27], q[27] );
    and ( up[29], up[28], q[28] );
    and ( up[30], up[29], q[29] );
    and ( up[31], up[30], q[30] );

    and ( up32, up[31], q[31] ); //needed for carry output

    //row of and gates used in the DOWN (DEC) instruction
    and ( down[1], ~up_down,   qnot[0] );
    and ( down[2],  down[1],   qnot[1] );
    and ( down[3],  down[2],   qnot[2] );
    and ( down[4],  down[3],   qnot[3] );
    and ( down[5],  down[4],   qnot[4] );
    and ( down[6],  down[5],   qnot[5] );
    and ( down[7],  down[6],   qnot[6] );
    and ( down[8],  down[7],   qnot[7] );
    and ( down[9],  down[8],   qnot[8] );
    and ( down[10], down[9],   qnot[9] );
    and ( down[11], down[10],  qnot[10] );
    and ( down[12], down[11],  qnot[11] );
    and ( down[13], down[12],  qnot[12] );
    and ( down[14], down[13],  qnot[13] );
    and ( down[15], down[14],  qnot[14] );
    and ( down[16], down[15],  qnot[15] );
    and ( down[17], down[16],  qnot[16] );
    and ( down[18], down[17],  qnot[17] );
    and ( down[19], down[18],  qnot[18] );
    and ( down[20], down[19],  qnot[19] );
    and ( down[21], down[20],  qnot[20] );
    and ( down[22], down[21],  qnot[21] );
    and ( down[23], down[22],  qnot[22] );
    and ( down[24], down[23],  qnot[23] );
    and ( down[25], down[24],  qnot[24] );
    and ( down[26], down[25],  qnot[25] );
    and ( down[27], down[26],  qnot[26] );
    and ( down[28], down[27],  qnot[27] );
    and ( down[29], down[28],  qnot[28] );
    and ( down[30], down[29],  qnot[29] );
    and ( down[31], down[30],  qnot[30] );

    and ( down32, down[31], qnot[31] ); //needed for carry output

    //or gates used in the count (INC/DEC) instruction
    or ( or_from[1],  down[1],   up[1], ones_comp );
```

```
    or ( or_from[2],  down[2],   up[2], ones_comp );
    or ( or_from[3],  down[3],   up[3], ones_comp );
    or ( or_from[4],  down[4],   up[4], ones_comp );
    or ( or_from[5],  down[5],   up[5], ones_comp );
    or ( or_from[6],  down[6],   up[6], ones_comp );
    or ( or_from[7],  down[7],   up[7], ones_comp );
    or ( or_from[8],  down[8],   up[8], ones_comp );
    or ( or_from[9],  down[9],   up[9], ones_comp );
    or ( or_from[10], down[10],  up[10], ones_comp );
    or ( or_from[11], down[11],  up[11], ones_comp );
    or ( or_from[12], down[12],  up[12], ones_comp );
    or ( or_from[13], down[13],  up[13], ones_comp );
    or ( or_from[14], down[14],  up[14], ones_comp );
    or ( or_from[15], down[15],  up[15], ones_comp );
    or ( or_from[16], down[16],  up[16], ones_comp );
    or ( or_from[17], down[17],  up[17], ones_comp );
    or ( or_from[18], down[18],  up[18], ones_comp );
    or ( or_from[19], down[19],  up[19], ones_comp );
    or ( or_from[20], down[20],  up[20], ones_comp );
    or ( or_from[21], down[21],  up[21], ones_comp );
    or ( or_from[22], down[22],  up[22], ones_comp );
    or ( or_from[23], down[23],  up[23], ones_comp );
    or ( or_from[24], down[24],  up[24], ones_comp );
    or ( or_from[25], down[25],  up[25], ones_comp );
    or ( or_from[26], down[26],  up[26], ones_comp );
    or ( or_from[27], down[27],  up[27], ones_comp );
    or ( or_from[28], down[28],  up[28], ones_comp );
    or ( or_from[29], down[29],  up[29], ones_comp );
    or ( or_from[30], down[30],  up[30], ones_comp );
    or ( or_from[31], down[31],  up[31], ones_comp );

    //build the carry output
    or ( or_from32, down32, up32 );
    and ( carry, or_from32, ~_reg_select, count );

    //build the overflow output
    //only an overflow of top bit toggles without lower bit
    and ( v1, qnot[31], or_from[31], q[30], ~_reg_select, count );
    and ( v2, q[31], or_from[31], qnot[30], ~_reg_select, count );
    or  ( overflow, v1, v2 );

    //build the negative output
    and ( negative, or_from[31], qnot[31], ~_reg_select, count );

    //build the zero output
    and ( z1,  q[0],  q[1],  q[2],  q[3],  q[4],  q[5],  q[6],  q[7],
               q[8],  q[9], q[10], q[11], q[12], q[13], q[14], q[15],
              q[16], q[17], q[18], q[19], q[20], q[21], q[22], q[23],
              q[24], q[25], q[26], q[27], q[28], q[29], q[30], q[31],
              ~_reg_select, up_down );

    and ( z2, q[0],  qnot[1],  qnot[2],  qnot[3],
           qnot[4],  qnot[5],  qnot[6],  qnot[7],
           qnot[8],  qnot[9], qnot[10], qnot[11],
```

```
        qnot[12], qnot[13], qnot[14], qnot[15],
        qnot[16], qnot[17], qnot[18], qnot[19],
        qnot[20], qnot[21], qnot[22], qnot[23],
        qnot[24], qnot[25], qnot[26], qnot[27],
        qnot[28], qnot[29], qnot[30], qnot[31],
                            ~_reg_select, ~up_down );
or ( zero, z1, z2 );

//row of and gates used to perform CLEAR instruction
and ( _clr[0],  this_d[0],  this_clear );
and ( _clr[1],  this_d[1],  this_clear );
and ( _clr[2],  this_d[2],  this_clear );
and ( _clr[3],  this_d[3],  this_clear );
and ( _clr[4],  this_d[4],  this_clear );
and ( _clr[5],  this_d[5],  this_clear );
and ( _clr[6],  this_d[6],  this_clear );
and ( _clr[7],  this_d[7],  this_clear );
and ( _clr[8],  this_d[8],  this_clear );
and ( _clr[9],  this_d[9],  this_clear );
and ( _clr[10], this_d[10], this_clear );
and ( _clr[11], this_d[11], this_clear );
and ( _clr[12], this_d[12], this_clear );
and ( _clr[13], this_d[13], this_clear );
and ( _clr[14], this_d[14], this_clear );
and ( _clr[15], this_d[15], this_clear );
and ( _clr[16], this_d[16], this_clear );
and ( _clr[17], this_d[17], this_clear );
and ( _clr[18], this_d[18], this_clear );
and ( _clr[19], this_d[19], this_clear );
and ( _clr[20], this_d[20], this_clear );
and ( _clr[21], this_d[21], this_clear );
and ( _clr[22], this_d[22], this_clear );
and ( _clr[23], this_d[23], this_clear );
and ( _clr[24], this_d[24], this_clear );
and ( _clr[25], this_d[25], this_clear );
and ( _clr[26], this_d[26], this_clear );
and ( _clr[27], this_d[27], this_clear );
and ( _clr[28], this_d[28], this_clear );
and ( _clr[29], this_d[29], this_clear );
and ( _clr[30], this_d[30], this_clear );
and ( _clr[31], this_d[31], this_clear );

//rows of or gates to load data
or ( _pre[0],  ~d[0],   this_load );
or ( _pre[1],  ~d[1],   this_load );
or ( _pre[2],  ~d[2],   this_load );
or ( _pre[3],  ~d[3],   this_load );
or ( _pre[4],  ~d[4],   this_load );
or ( _pre[5],  ~d[5],   this_load );
or ( _pre[6],  ~d[6],   this_load );
or ( _pre[7],  ~d[7],   this_load );
or ( _pre[8],  ~d[8],   this_load );
or ( _pre[9],  ~d[9],   this_load );
or ( _pre[10], ~d[10],  this_load );
```

```
    or ( _pre[11], ~d[11],  this_load );
    or ( _pre[12], ~d[12],  this_load );
    or ( _pre[13], ~d[13],  this_load );
    or ( _pre[14], ~d[14],  this_load );
    or ( _pre[15], ~d[15],  this_load );
    or ( _pre[16], ~d[16],  this_load );
    or ( _pre[17], ~d[17],  this_load );
    or ( _pre[18], ~d[18],  this_load );
    or ( _pre[19], ~d[19],  this_load );
    or ( _pre[20], ~d[20],  this_load );
    or ( _pre[21], ~d[21],  this_load );
    or ( _pre[22], ~d[22],  this_load );
    or ( _pre[23], ~d[23],  this_load );
    or ( _pre[24], ~d[24],  this_load );
    or ( _pre[25], ~d[25],  this_load );
    or ( _pre[26], ~d[26],  this_load );
    or ( _pre[27], ~d[27],  this_load );
    or ( _pre[28], ~d[28],  this_load );
    or ( _pre[29], ~d[29],  this_load );
    or ( _pre[30], ~d[30],  this_load );
    or ( _pre[31], ~d[31],  this_load );

    or ( this_d[0],  d[0],   this_load );
    or ( this_d[1],  d[1],   this_load );
    or ( this_d[2],  d[2],   this_load );
    or ( this_d[3],  d[3],   this_load );
    or ( this_d[4],  d[4],   this_load );
    or ( this_d[5],  d[5],   this_load );
    or ( this_d[6],  d[6],   this_load );
    or ( this_d[7],  d[7],   this_load );
    or ( this_d[8],  d[8],   this_load );
    or ( this_d[9],  d[9],   this_load );
    or ( this_d[10], d[10],  this_load );
    or ( this_d[11], d[11],  this_load );
    or ( this_d[12], d[12],  this_load );
    or ( this_d[13], d[13],  this_load );
    or ( this_d[14], d[14],  this_load );
    or ( this_d[15], d[15],  this_load );
    or ( this_d[16], d[16],  this_load );
    or ( this_d[17], d[17],  this_load );
    or ( this_d[18], d[18],  this_load );
    or ( this_d[19], d[19],  this_load );
    or ( this_d[20], d[20],  this_load );
    or ( this_d[21], d[21],  this_load );
    or ( this_d[22], d[22],  this_load );
    or ( this_d[23], d[23],  this_load );
    or ( this_d[24], d[24],  this_load );
    or ( this_d[25], d[25],  this_load );
    or ( this_d[26], d[26],  this_load );
    or ( this_d[27], d[27],  this_load );
    or ( this_d[28], d[28],  this_load );
    or ( this_d[29], d[29],  this_load );
    or ( this_d[30], d[30],  this_load );
    or ( this_d[31], d[31],  this_load );
```

```
endmodule
```

Chapter 5

Arithmetic and Logical Calculations Units

The Pioneer-2 processor has several computational units, which are responsible for performing calculations. These units receive their inputs from the "X" and "Y" busses, and place their results onto the "result" bus. The computational units all may be effected by the status of the carry flag, and the process of performing a calculation may effect the flags as part of the result. The destination register will always be the same register as the first argument in a computational statement. For example, the statement:

```
R5 = R5 + R12 ;Add R12 to R5
```

would be a legal instruction, while the statements:

```
R5 = R8 + R12 ;Add R12 to R8, store in R5
```

```
R5 = R12 + R5 ;Add R12 to R5
```

would not be legal instructions.

The computational units include the Accumulator/Logic Unit (ALU), and shifter, and the multiplier/divider[1]. Each of these computational units acts independantly of the other units. This will allow some parallelism inside the processor because some processes (i.e. division) take longer than other processes (addition)[2].

The input values of each computational unit are latched, so once the computational unit has started it's process, the "X" and "Y" busses are fre to be changed for oher pipelined events. Likewise, the output values of each computational unit are latched, and the outputs to the "result" bus can be left floating until the CPU controller enables the output from the computational unit onto the "result" bus.

Each of the computational units shares a set of control and status signals with the CPU controller. When the controller initiates a calculation, is sends a signal to the computation unit, indicating that the data on the "X" and "Y" busses is for that unit, it should latch those values, and begin it's calculation. This process sets a flag indicating that the computational unit is full (in use), and another flag to indicate that it is performing a calculation. Once the computational unit has completed it's calculation, it clears the flag that it is performing a calculation, indicating to the CPU controller that the data is ready. The CPU controller can then read the data from the output of the computational unit, placing the data on the "result" bus, then clearing the full flag, indicating that the computational unit is ready for a new operation.

[1] Although the multiplier and the divider are seperate units, they are typically collected together because of their similar functions.

[2] The capability of allowing a multiply to operate and still allow addition in the ALU to occur in parallel may (or may not) be included.

5.1 ALU - Arithmetic Logic Unit

The ALU is the unit that is responsible for adding, subtracting, and comparing data, as well as performing most of the logical functions such as AND, OR, XOR, and NOT. For the arithmetic functions (addition, subtraction, and comparison), the data is assumed to be signed. Instructions may be modified to indicate that the data is unsigned. Signedness does not apply to the logical functions.

Any ALU instructions has at least one parameter; most have two. These parameters are placed on the "X" and "Y" busses insinde the CPU. (Single-parameter instructions only use the "X" bus.) Once these values are on the bus, the ALU receives instructions from the CPU controller, instructing which calculation to perform, and the output is calculated.

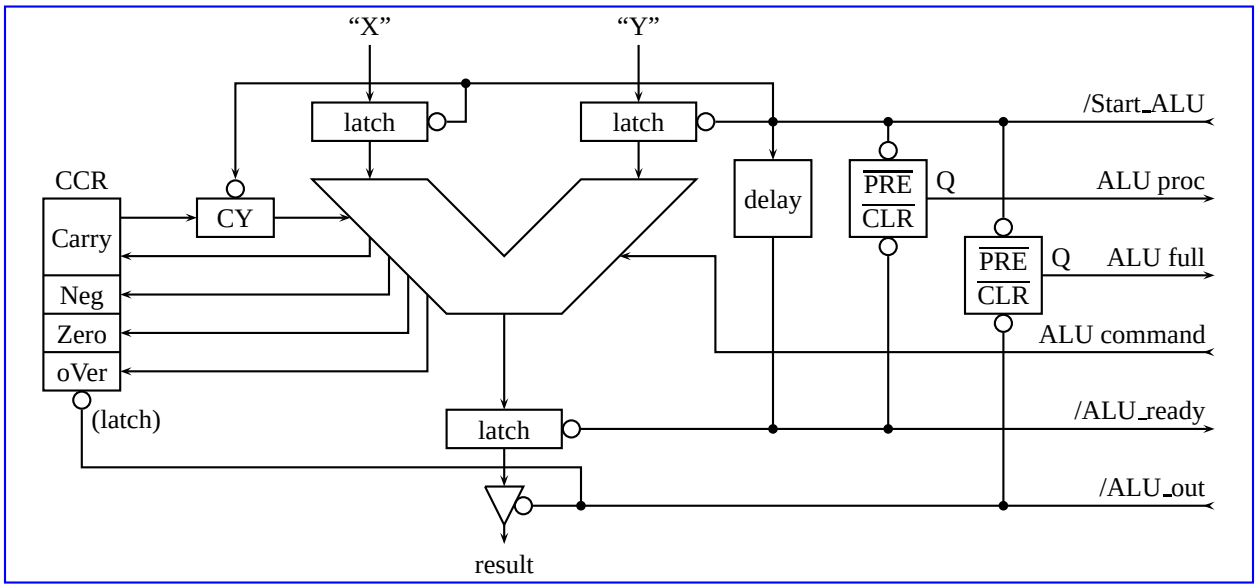


Figure 5.1: ALU

The ALU is selected when instruction$_{27,26}$ bits are both 0. Instruction$_{23-25}$ bits are used to select which instruction is to be performed by the ALU, according to Table 5.1.

Instr$_{25}$	Instr$_{24}$	Instr$_{23}$	ALU Operation	(un)Signed	Invertable	Brief
0	0	0	AND		Y	Logical "AND" function
0	0	1	OR		Y	Logical "OR" function
0	1	0	XOR		Y	Logical "XOR" (exclusive-or) function
0	1	1	PASS		Y	Pass value through (set flags) function
1	0	0	ADD	Y		Arithmetic addition
1	0	1	SUB	Y		Arithmetic subtraction
1	1	0	NEG	ignored		2's compliment
1	1	1	CMP	Y		Perform subtraction, set flags only

Table 5.1: ALU operations

5.1.1 Logical Function Descriptions

The logical functions are very flexible. There are four primary functions, each of which may be inverted (by setting instruction$_{22}$ to 1) to invert the output of the ALU for logical operations.

AND performs a logical “AND” function for each pair of bits between the two parameters. If the output is inverted, the logical function “NAND” will be the result.

OR performes a logical “OR” function for each pair of bits between the two parameters. If the output is inverted, the logical function “NOR” will be performed.

XOR performs a logical “XOR” function for each pair of bits between the two arguements. If the output is inverted, the logical function “XNOR” will be performed.

PASS performs no operation on the single parameter, just passes it through the ALU so that sign flag and/or zero flag can be checked. If inverted, the logical operation “NOT” (1’s compliment) will be performed.

5.1.2 Arithmetic Function Descriptions

The arithmetic funnctions are limited to addition and subtraction. (Other arithmetic functions are handled elsewhere.)

ADD performs an arithmetic additions between two values. If the result of an addition exceeds 2147483647 ($2^{31}-1$), the overflow flag is set, and the carry is cleared. If a sum greater than 4294967295 ($2^{32}-1$) is reached, the carry flag is set. All ADD instructions consider the carry flag as an input, so if the Carry Flag is set, the result of the ADD instruction will have an additional one added.

SUB performs an arithmetic subtraction between two values. If a subtraction ends with a result less than -2147483648 (-2^{31}), the overflow flag is set, otherwise, it is cleared. If a subtraction results in a value less than 0, the carry flag is cleared, but is otherwise set. All SUB instructions include the Carry Flag as an input.

NEG will perform a 2’s compliment of the parameter. In the unique case where the number -2147483648 is being negated, the carry and overflow flags will be set; otherwise they will always be cleared after a NEG instruction.

CMP performs the same operations as SUB except that the result value is not written back into any register. The ALU operation is exactly the same, but the CPU controller will not save the resulting differance into any registers. ONLY the carry, negative, overflow, and zero flags will be set.

5.1.3 Immediate Data

Any of the ALU commands may have a 16-bit value stored on the “Y” bus. This is done when instruction$_{21}$ is set. (NOTE: the “PASS” and “NOT” instructions will ignore data on the “Y” bus, and the “NEG” instruction will not allow immediate data onto the “Y” bus.) Using this, immediate data may be the second parameter to any of the functions. If the operation’s instruction$_{22}$ is set, the input value will be sign-extended when they are placed on the “Y” bus. Sign-extension will occur for arithmetic instructions or for logical instructions.

5.1.4 Instruction Set for the ALU

The instruction set for the ALU is designed to appear as arithmetic equations. The operators, + and - denote addition and subtraction, and the logical operators are defined using the anme of the operation, “AND,” “OR,” “XOR,[3] and “NOT.” The following examples show each of the ALU operations:

[3] “EOR” may also be used as a synonym for “XOR.”

```
R8 = R8 AND R23     ;‘‘AND’’ operation
R5 = R5 NAND R2     ;‘‘NAND’’ operation
R5 =!  R5 AND R18   ;‘‘NAND’’ operation (alternate form)
R12 = R12 OR R5     ;‘‘OR’’ operation
R5 = R5 NOR R23     ;‘‘NOR’’ operation
R23 =!  R23 OR R12  ;‘‘NOR’’ operation (alternate form)
R7 = R7 XOR R18     ;‘‘exclusive-OR’’ operation
R4 = R4 XNOR R25    ;‘‘exclusive-NOR’’ operation
R0 =!  R0 XOR R6    ;‘‘exclusive-NOR’’ operation (alternate form)
R11 = R11           ;‘‘PASS’’ operation
R25 = NOT R25       ;‘‘NOT’’ (one’s compliment) operation
R12 =!  R12         ;‘‘NOT’’ (one’s compliment) operation (alternate form)
R8 = R8 + R4        ;Addition
R12 = R12 - R25     ;Subtraction
R0 = -R0            ;‘‘NEG’’ (two-s compliment) operation
R25 - R23           ;‘‘CMP’’ operation (no destination register)
CHECK R25 - R23     ;‘‘CMP’’ operation (alternate form)
```

5.1.5 ALU Code

The verilog module for the ALU is found in the file, `alu.v`:

```
module alu( result, c_out, n_out, v_out, z_out, x, y, cmd, sign_mode, c_in );
  output [31:0] result;
  output        c_out;
  output        n_out;
  output        v_out;
  output        z_out;
  input  [31:0] x,y;
  input  [2:0]  cmd;
  input         sign_mode;
  input         c_in;

  wire [31:0] adder_out;
  wire [31:0] and_out;
  wire [31:0] or_out;
  wire [31:0] xor_out;
  wire [31:0] pre_out;
  wire [31:0] adder_x; //wires feed x input into adder,
                       //may be cleared for NEG
  wire [31:0] adder_y; //wires feed y input into adder,
                       //may be inverted for SUB, NEG, CMP
  wire        clear_x;

  and ( and_out[0], x[0], y[0]);
  and ( and_out[1], x[1], y[1]);
  and ( and_out[2], x[2], y[2]);
  and ( and_out[3], x[3], y[3]);
  and ( and_out[4], x[4], y[4]);
  and ( and_out[5], x[5], y[5]);
  and ( and_out[6], x[6], y[6]);
```

```
    and ( and_out[7], x[7], y[7]);
    and ( and_out[8], x[8], y[8]);
    and ( and_out[9], x[9], y[9]);
    and (and_out[10],x[10],y[10]);
    and (and_out[11],x[11],y[11]);
    and (and_out[12],x[12],y[12]);
    and (and_out[13],x[13],y[13]);
    and (and_out[14],x[14],y[14]);
    and (and_out[15],x[15],y[15]);
    and (and_out[16],x[16],y[16]);
    and (and_out[17],x[17],y[17]);
    and (and_out[18],x[18],y[18]);
    and (and_out[19],x[19],y[19]);
    and (and_out[20],x[20],y[20]);
    and (and_out[21],x[21],y[21]);
    and (and_out[22],x[22],y[22]);
    and (and_out[23],x[23],y[23]);
    and (and_out[24],x[24],y[24]);
    and (and_out[25],x[25],y[25]);
    and (and_out[26],x[26],y[26]);
    and (and_out[27],x[27],y[27]);
    and (and_out[28],x[28],y[28]);
    and (and_out[29],x[29],y[29]);
    and (and_out[30],x[30],y[30]);
    and (and_out[31],x[31],y[31]);

    or ( or_out[0], x[0], y[0]);
    or ( or_out[1], x[1], y[1]);
    or ( or_out[2], x[2], y[2]);
    or ( or_out[3], x[3], y[3]);
    or ( or_out[4], x[4], y[4]);
    or ( or_out[5], x[5], y[5]);
    or ( or_out[6], x[6], y[6]);
    or ( or_out[7], x[7], y[7]);
    or ( or_out[8], x[8], y[8]);
    or ( or_out[9], x[9], y[9]);
    or (or_out[10],x[10],y[10]);
    or (or_out[11],x[11],y[11]);
    or (or_out[12],x[12],y[12]);
    or (or_out[13],x[13],y[13]);
    or (or_out[14],x[14],y[14]);
    or (or_out[15],x[15],y[15]);
    or (or_out[16],x[16],y[16]);
    or (or_out[17],x[17],y[17]);
    or (or_out[18],x[18],y[18]);
    or (or_out[19],x[19],y[19]);
    or (or_out[20],x[20],y[20]);
    or (or_out[21],x[21],y[21]);
    or (or_out[22],x[22],y[22]);
    or (or_out[23],x[23],y[23]);
    or (or_out[24],x[24],y[24]);
    or (or_out[25],x[25],y[25]);
    or (or_out[26],x[26],y[26]);
    or (or_out[27],x[27],y[27]);
```

```
    or (or_out[28],x[28],y[28]);
    or (or_out[29],x[29],y[29]);
    or (or_out[30],x[30],y[30]);
    or (or_out[31],x[31],y[31]);

    xor ( xor_out[0], x[0], y[0]);
    xor ( xor_out[1], x[1], y[1]);
    xor ( xor_out[2], x[2], y[2]);
    xor ( xor_out[3], x[3], y[3]);
    xor ( xor_out[4], x[4], y[4]);
    xor ( xor_out[5], x[5], y[5]);
    xor ( xor_out[6], x[6], y[6]);
    xor ( xor_out[7], x[7], y[7]);
    xor ( xor_out[8], x[8], y[8]);
    xor ( xor_out[9], x[9], y[9]);
    xor (xor_out[10],x[10],y[10]);
    xor (xor_out[11],x[11],y[11]);
    xor (xor_out[12],x[12],y[12]);
    xor (xor_out[13],x[13],y[13]);
    xor (xor_out[14],x[14],y[14]);
    xor (xor_out[15],x[15],y[15]);
    xor (xor_out[16],x[16],y[16]);
    xor (xor_out[17],x[17],y[17]);
    xor (xor_out[18],x[18],y[18]);
    xor (xor_out[19],x[19],y[19]);
    xor (xor_out[20],x[20],y[20]);
    xor (xor_out[21],x[21],y[21]);
    xor (xor_out[22],x[22],y[22]);
    xor (xor_out[23],x[23],y[23]);
    xor (xor_out[24],x[24],y[24]);
    xor (xor_out[25],x[25],y[25]);
    xor (xor_out[26],x[26],y[26]);
    xor (xor_out[27],x[27],y[27]);
    xor (xor_out[28],x[28],y[28]);
    xor (xor_out[29],x[29],y[29]);
    xor (xor_out[30],x[30],y[30]);
    xor (xor_out[31],x[31],y[31]);

    or( the_sign, cmd[1], cmd[2] );
    nand( clear_x, cmd[2], cmd[1], ~cmd[0] );

    and ( adder_x[0], clear_x, x[0]);
    and ( adder_x[1], clear_x, x[1]);
    and ( adder_x[2], clear_x, x[2]);
    and ( adder_x[3], clear_x, x[3]);
    and ( adder_x[4], clear_x, x[4]);
    and ( adder_x[5], clear_x, x[5]);
    and ( adder_x[6], clear_x, x[6]);
    and ( adder_x[7], clear_x, x[7]);
    and ( adder_x[8], clear_x, x[8]);
    and ( adder_x[9], clear_x, x[9]);
    and (adder_x[10], clear_x,x[10]);
    and (adder_x[11], clear_x,x[11]);
    and (adder_x[12], clear_x,x[12]);
```

```
    and (adder_x[13], clear_x,x[13]);
    and (adder_x[14], clear_x,x[14]);
    and (adder_x[15], clear_x,x[15]);
    and (adder_x[16], clear_x,x[16]);
    and (adder_x[17], clear_x,x[17]);
    and (adder_x[18], clear_x,x[18]);
    and (adder_x[19], clear_x,x[19]);
    and (adder_x[20], clear_x,x[20]);
    and (adder_x[21], clear_x,x[21]);
    and (adder_x[22], clear_x,x[22]);
    and (adder_x[23], clear_x,x[23]);
    and (adder_x[24], clear_x,x[24]);
    and (adder_x[25], clear_x,x[25]);
    and (adder_x[26], clear_x,x[26]);
    and (adder_x[27], clear_x,x[27]);
    and (adder_x[28], clear_x,x[28]);
    and (adder_x[29], clear_x,x[29]);
    and (adder_x[30], clear_x,x[30]);
    and (adder_x[31], clear_x,x[31]);

    or  (subtraction, cmd[0], cmd[1]);
    nand (inv_y, subtraction, cmd[2]);

    xor ( adder_y[0], inv_y, y[0]);
    xor ( adder_y[1], inv_y, y[1]);
    xor ( adder_y[2], inv_y, y[2]);
    xor ( adder_y[3], inv_y, y[3]);
    xor ( adder_y[4], inv_y, y[4]);
    xor ( adder_y[5], inv_y, y[5]);
    xor ( adder_y[6], inv_y, y[6]);
    xor ( adder_y[7], inv_y, y[7]);
    xor ( adder_y[8], inv_y, y[8]);
    xor ( adder_y[9], inv_y, y[9]);
    xor (adder_y[10], inv_y,y[10]);
    xor (adder_y[11], inv_y,y[11]);
    xor (adder_y[12], inv_y,y[12]);
    xor (adder_y[13], inv_y,y[13]);
    xor (adder_y[14], inv_y,y[14]);
    xor (adder_y[15], inv_y,y[15]);
    xor (adder_y[16], inv_y,y[16]);
    xor (adder_y[17], inv_y,y[17]);
    xor (adder_y[18], inv_y,y[18]);
    xor (adder_y[19], inv_y,y[19]);
    xor (adder_y[20], inv_y,y[20]);
    xor (adder_y[21], inv_y,y[21]);
    xor (adder_y[22], inv_y,y[22]);
    xor (adder_y[23], inv_y,y[23]);
    xor (adder_y[24], inv_y,y[24]);
    xor (adder_y[25], inv_y,y[25]);
    xor (adder_y[26], inv_y,y[26]);
    xor (adder_y[27], inv_y,y[27]);
    xor (adder_y[28], inv_y,y[28]);
    xor (adder_y[29], inv_y,y[29]);
    xor (adder_y[30], inv_y,y[30]);
```

```
    xor (adder_y[31], inv_y,y[31]);

    xor (adder_c_in, c_in, inv_y);

    fast_addsub32 adder( adder_out, adder_c_out,
                         adder_x, y, c_in, subtraction );
    //only have carry out of add/sub/cmp/neg
    and( c_out, adder_c_out, cmd[2] );

    select_one_of_eight32 alu_sel( pre_out, and_out, or_out, xor_out, y,
                         adder_out, adder_out, adder_out, adder_out, cmd );

    and (logical_inv,~cmd[2],sign_mode);

    xor ( result[0], pre_out[0],logical_inv);
    xor ( result[1], pre_out[1],logical_inv);
    xor ( result[2], pre_out[2],logical_inv);
    xor ( result[3], pre_out[3],logical_inv);
    xor ( result[4], pre_out[4],logical_inv);
    xor ( result[5], pre_out[5],logical_inv);
    xor ( result[6], pre_out[6],logical_inv);
    xor ( result[7], pre_out[7],logical_inv);
    xor ( result[8], pre_out[8],logical_inv);
    xor ( result[9], pre_out[9],logical_inv);
    xor (result[10],pre_out[10],logical_inv);
    xor (result[11],pre_out[11],logical_inv);
    xor (result[12],pre_out[12],logical_inv);
    xor (result[13],pre_out[13],logical_inv);
    xor (result[14],pre_out[14],logical_inv);
    xor (result[15],pre_out[15],logical_inv);
    xor (result[16],pre_out[16],logical_inv);
    xor (result[17],pre_out[17],logical_inv);
    xor (result[18],pre_out[18],logical_inv);
    xor (result[19],pre_out[19],logical_inv);
    xor (result[20],pre_out[20],logical_inv);
    xor (result[21],pre_out[21],logical_inv);
    xor (result[22],pre_out[22],logical_inv);
    xor (result[23],pre_out[23],logical_inv);
    xor (result[24],pre_out[24],logical_inv);
    xor (result[25],pre_out[25],logical_inv);
    xor (result[26],pre_out[26],logical_inv);
    xor (result[27],pre_out[27],logical_inv);
    xor (result[28],pre_out[28],logical_inv);
    xor (result[29],pre_out[29],logical_inv);
    xor (result[30],pre_out[30],logical_inv);
    xor (result[31],pre_out[31],logical_inv);

    buf (n_out,result[31]);

    nor (z_out, result[0], result[1], result[2], result[3],
                result[4], result[5], result[6], result[7],
                result[8], result[9],result[10],result[11],
               result[12],result[13],result[14],result[15],
               result[16],result[17],result[18],result[19],
```

```
            result[20],result[21],result[22],result[23],
            result[24],result[25],result[26],result[27],
            result[28],result[29],result[30],result[31] );

  and (overflow,cmd[2],~cmd[1],~cmd[0],~x[31],result[31]);
  and (underflow,cmd[2],subtraction,x[31],~result[31]);
  or  (v_out,overflow,underflow);
endmodule
```

5.2 Shifter Logic Unit

The shifter unit performs all shifting and rotating functions. Instead of a simple 1-bit shifter, it is a barrel shifter, which means it can shift multiple spaces, according to a second input, which comes from the "Y" internal bus. It can perform several different functions, according to the input command. It can perform a logical shift in either direction, and a right-shift operations can be switched to an arithmetic shift by setting the sign bit, instruction$_{22}$. A left shift will ignore this sign bit, but a right shift will sign extend the MSB as the shift occurs.

Shifter instructions always have two parameters. The first parameter is the value to be shifted, which is placed on the "X" bus. The second parameter is the shift distance, and is placed on the "Y" bus. The second parameter may be an immediate value, and this will always refer to the shift amount.

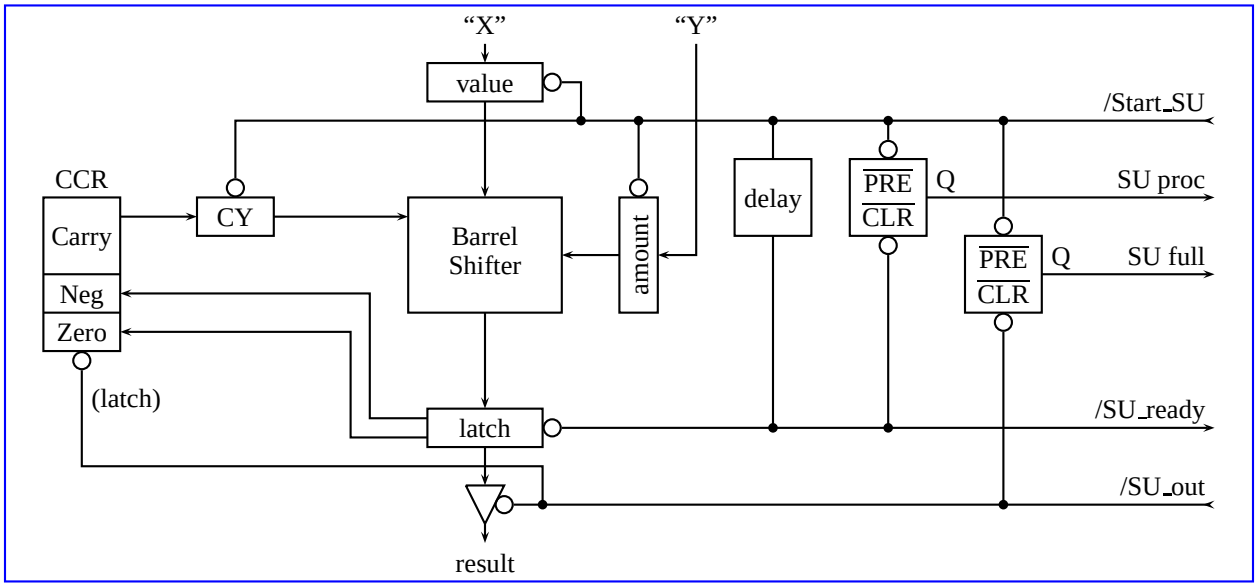


Figure 5.2: Shifter

The shifter is selected when the instruction$_{27,26,25}$ bits are all 0. Instruction$_{24,23}$ bits are used to select which shift instruction is to be performed, and the sign bit (instruction$_{22}$) is used to modify the function of the shift instruction.

$Instr_{24}$	$Instr_{23}$	SU Operation	(un)Signed	Brief
0	0	LSHIFT	ignored	Logical shift left function
0	1	RSHIFT	If 1, ASHIFT	Logical or Arithmetic shift right
1	0	LROTATE	If 1, LROTATEC	Logical rotate left (through carry?) function
1	1	RROTATE	If 1, RROTATEC	Logical rotate right (through carry?) function

Table 5.2: Shifter Operations

5.2.1 Logical Shift

In a logical shift, the input value (from the "X" bus) will be shifted a specific number of bits, controlled by the second parameter (on the "Y" bus). The value on the "Y" bus is assumed to be an unsigned (positive) value because the direction of the shift will be controlled by the selection of the instruction.

LSHIFT will perform a left (upward) shift. As bits are shifted to the left, the highest bit falls into the carry flag, and the previous values of the carry flag are discarded. The LSB is filled with 0s as data to be shifted in.

RSHIFT will perform a right (downward) shift. As bits are shifted to the right, 0s are shifted in to the MSB, and the LSB is shifted into the carry flag with the previous value of the carry flag being discarded.

5.2.2 Arithmetic Shift

ASHIFT will perform a right shift, similar to RSHIFT, but the sign bit (bit 31) is kept, and this value is used to refill bit 31 after the shift, instead of the 0 that would have been inserted here from an RSHIFT instruction. The instruction encoding for ASHIFT is the same as that for RSHIFT, except that the sign bit ($instruction_{22}$) is set.

Because the left-most bit (highest order bit) is the sign bit, a shift to the left would always over-write the data in the sign bit. Therefore, there is no LASHIFT instruction. Setting the $instruction_{22}$ bit is ignored.

5.2.3 Rotate

There are two rotate instructions, LROTATE and RROTATE. These instructions behave very simlarly to the shift instructions, eccept that for each shift, where data is shifted into the carry flag, then discarded, this data is rotated back into the bit that was vacated by the shift. The rotate may occur by rotating data past the cary flag, or through the carry flag. If rotating past the carry flag, the data that was shifted into the carry flag is also the data being shifted in to the opposite end of the register. If rotating through the carry flag, the data shifts out of the register, into the carry, and the carry flag's value that is discarded in a shift is placed into the opposite end of the register.

Encoding the rotate instruction uses the sign bit ($instruction_{22}$) to select between rotating through the carry (bit=1) or by the carry (bit=0).

Figure 5.3 demonstrates the different modes of shifting that is performed in the shifter.

Do I want to include a Detect Exponent function? I think it would be pretty easy, but it's also another one of those things that is used on fixed-point or floating point processors.

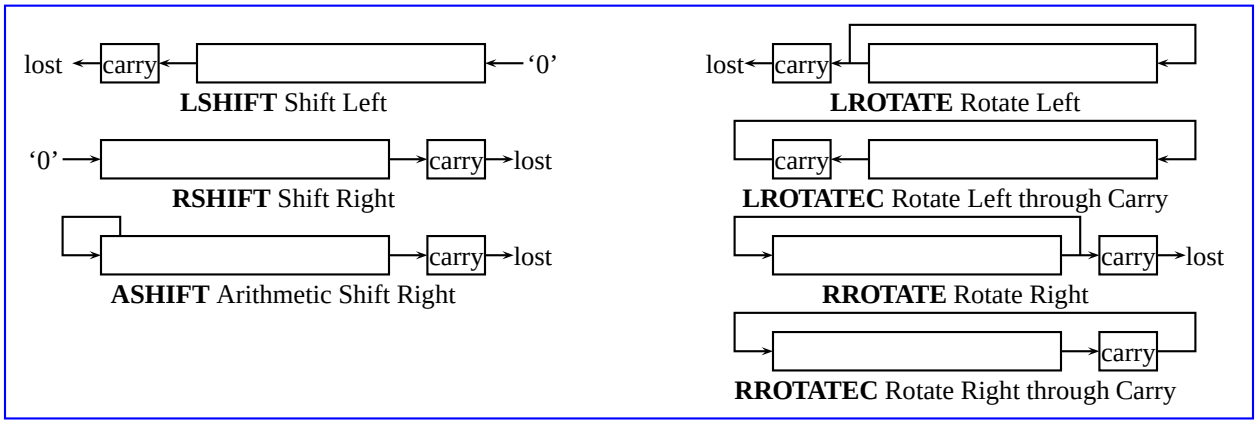


Figure 5.3: Shift Modes

5.2.4 Instruction Set of the Shifter Unit

The shifter unit receives two arguments in each operation. The first argument is the value to be shifted, and the second operation is always the value to shift the value by. The second argument is always an unsigned value, and the low 5 bits are always mased as the shift value. If the second arguement had the actual value of 32 (low 5 bits all 0), the effective shifted amount would be shifted by 0. The following example shows each of the shifter operations:

```
R2 = R2 LSHIFT BY R4      ;Logical left shift
R3 = R3 RSHIFT BY R4      ;Logical right shift
R6 = R6 ASHIFT BY R3      ;Arithmetic right shift
R24 = R24 LROTATE BY R2   ;Rotate left
R15 = R15 RROTATE BY R3   ;Rotate right
R24 = R24 LROTATEC BY R8  ;Rotate left through carry flag
R25 = R25 RROTATEC BY R9  ;Rotate right through carry flag

R8 = R8 ASHIFT BY 4       ;Arithmetic right shift of immediate value
R12 = R12 LSHIFT BY -6    ;Negative values not allowed
```

5.2.5 Shifter Code

The verilog code for the shifter can be found in the file `shifter.v`.

```
module shifter( result, x_in, y_in, mode, sign );
  output [31:0] result; //output result
  input  signed [31:0] x_in;   //value to shift
  input  [31:0]  y_in;   //amount to shift
  input  [1:0]  mode;   //only some of the modes are included for this
  version
  input         sign;   //1=Arith shift or Rotate through carry

  reg signed [31:0] result;

  reg [4:0] distance;
```

```
  always @(x_in or y_in or mode or sign )
    begin
      assign distance = y_in[4:0];
      case ( mode )
        //LSHIFT
        0 : result = x_in << distance;

        //RSHIFT
        1 : result = (sign==1'b1) ? x_in >>> distance : x_in >> distance;

        //LROTATE
        2 : result = ( x_in << distance ) | ( x_in >> (32-distance) );

        //RROTATE
        3 : result = ( x_in >> distance ) | ( x_in << (32-distance) );

        default :
        begin
          result=32'hffffffff;
          result[4]=y_in[4];
          result[3]=y_in[3];
          result[2]=y_in[2];
          result[1]=y_in[1];
          result[0]=y_in[0];
        end
      endcase
  end
endmodule
```

It is not complete, yet, and it is still modeled in behavioral model. The "rotate through carry" instructions are still incomplete, and the carry flag output is not finished.

5.3 Multiplier Logic Unit

The multpilier unit performs 32-bit by 32-bit multiplication. As a result, the answer is capable of being as large as 64-bits in size. Because the "results" bus is only 32 bits wide, only the low-order 32 bits are placed on the bus as the result of a multiplication. If a multiplication is performed which will require the addutional 32 bits, the overflow flag is set after the calculation, and the program may read the high-order 32 bits out of the MUL_HIGH special-purpose register.

5.3.1 Instruction Set for the Multiplier

The Multiplier is selected when instruction$_{27-23}$ form the binary value `0b01100`. Instruction$_{21}$ controls the immediate addressing mode, the same as the ALU and Shifter units. The instructions for multiplication appear very similar to the instructions for the ALU:

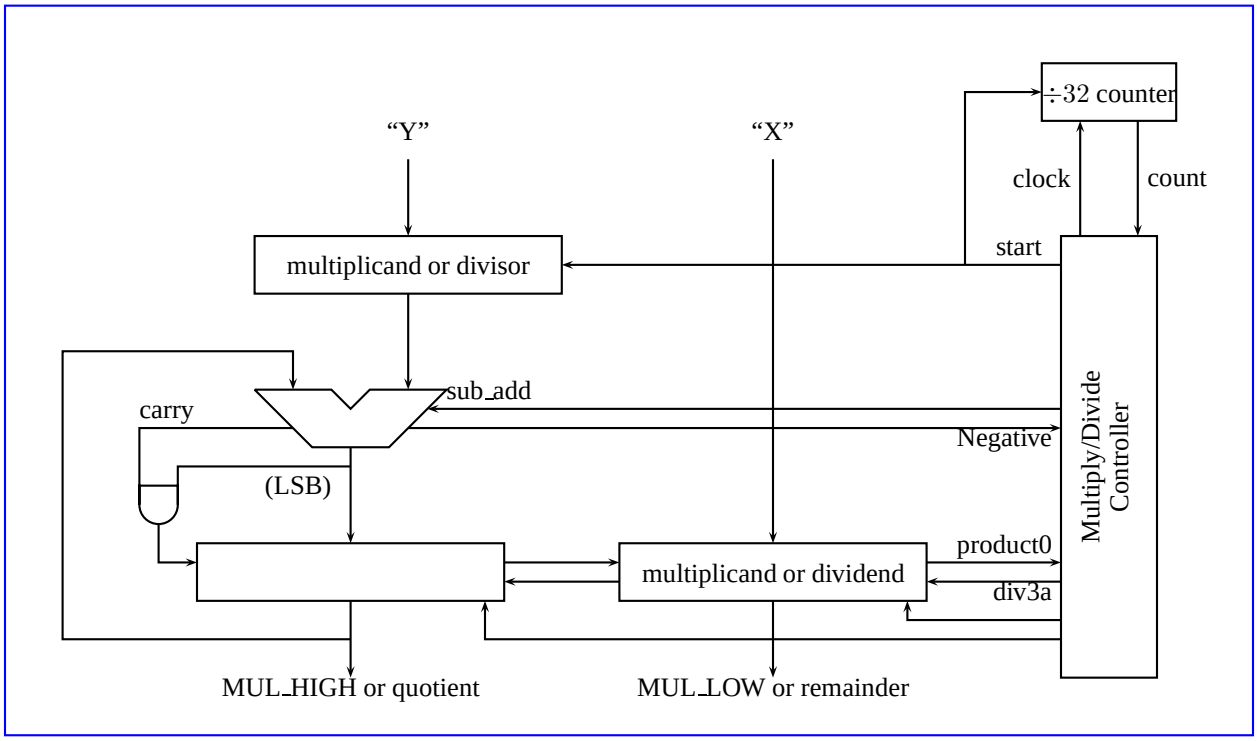


Figure 5.4: Multiplier

```
R2 = R2 * R4  ;Multiply R2 by R4, result in R2

R8 = R8 * 23  ;Multiply R8 by the immediate number 23
```

The result of two 32-bit factors could be a 64-bit result. When the result is moved into the result register, only the low 32 bits are copied. The high 32 bits are stored in the MUL_HIGH register, which can only be read as a special purpose register.

5.3.2 Multiplier Code

The verilog code for the multiplier can be found in the file `u_mul_div.v`:

```
module times( high, low, x, y );
  output [31:0] high, low;
  input  [31:0] x, y;

  reg [63:0] result;
  reg [31:0] high, low;

  always @( x or y )
    begin
      assign result=x*y;
      assign high=result[63:32];
      assign low=result[31:0];
```

```
    end
endmodule
```

Can I use the multiplier that is built into the S3E FPGA? It is only 16-bits (18bits?), but instead of doing 8 32x4-bit multiplies, maybe I can do 4 16x16-bit multiplies and add them (shifting appropriately)?

5.3.3 Multiply and Accumulate

I think I'll throw this out - it's great for DSP stuff, but those functions are operating in floating-point and/or fixed-point mode, and this project is only operating in integer-modes. I'm kinda really bummed about that, because it's not much more to add, but????????

5.4 Divider Logic Unit

The divider unit uses the same hardware as the multiplier, includeing the same pair of 32-bit registers, MUL_HIGH and MUL_LOW. The quotient is held in the MUL_HIGH register and the remainder is held in the MUL_LOW register after the division is completed.

5.4.1 Instruction Set for the Divider

The divider is selected when instruction$_{27-23}$ form the binary value `0b01110`, and instruction$_{22}$ controls the immediate addressing mode. Like the multiplier, the divider's instruction set has the same appearance as the ALU:

```
R3 = R3 / R0    ;Divide R3 by R0, result in R0

R18 = R18 / 23  ;Divide R18 by the immediate number 23
```

The result of the division provides both a quotient ("result"), and a remainder. The value stored in the result register is always the quotient. The remainder is always stored in the MUL_LOW register, which can only be accessed as a special purpose register.

5.4.2 Divider Code

The verilog module for the divider can be found in the same file as the multiplier, `u_mul_div.v`:

```
module div( dividend, remainder, x, y );
  output [31:0] dividend, remainder;
  input  [31:0] x, y;

  reg [31:0] dividend, remainder;
```

```
  always @( x or y )
    begin
      assign dividend = x/y;
      assign remainder = x-(dividend*y);
    end
endmodule
```

5.5 Multipler/Divider Integration

I plan to include the multiplier/divider unit that was discussed in class, using the same hardware for both functions, particularly if the S3E FPGA's multiplier doesn't work the way I'd like.

For this version of the Pioneer-2, the multiplier and divider needed to be re-integrated into the same large unit, so one last module had to be included in the u_mul_div.v file:

```
module mul_div( high, low, x, y, mode );
  output [31:0] high, low;
  input  [31:0] x, y;
  input  [1:0]  mode;

  wire [31:0] mul_high, mul_low, div_high, div_low;

  times mul   ( mul_high, mul_low, x, y );
  div   divid ( div_high, div_low, x, y );

  select_one_of_two32 highsel ( high, mul_high, div_high, mode[1] );
  select_one_of_two32 lowsel  ( low,  mul_low,  div_low,  mode[1] );
endmodule
```

This module acts as the superstructure for the multiplier/divider. It is written in the behavioral mode, and will be completely replaced in future versions.

Chapter 6

Address Generator

The Address Generator is a specially designed computational unit to perform memory indexing and address calculations. It has three sets of registers that are used for the calculations of memory addresses.

Figure 6.1 shows the functionality of the Address Generator. The output of a selected address register is added to a selected modify register, and the output from the length register performs masks to limit the ranges of the registers[1].

The first set of registers is the address register set (A0-A7). These registers contain the current pointers to memory. These registers can be modified automatically using other registers. The A7 register has the special purpose of being the stack-pointer.

The second set of registers is the modify register set (M0-M7). These registers contain signed values that are used to modify the address registers. These registers contain signed numbers, so both positive and negative adjustments can be made. Three registers contain hard-wired modify values. M0 contains the number 0, meaning make no adjustment. M1 contains the number +1, and M7 contains the number -1.

The third set of registers is the length register set (L0-L7). These registers contain values that define the lengths or boundaries for memory areas that limit the range of the address registers. The length register can be used to reset a modified register back to the beginning of an area if it exceeds the area after being modified. This can be useful for ring bufferswher reaching then end of the buffer will automatically reset back to the beginning. A length register is always selected with it's matching address register. Using the A0 register will always use the L0 register. If a length register has the value 0, this feature is ignored.

Example: If the A4 register has the value 0x80000000, L4 has the value 0x00000008 (length of 8), and M6 has the value +5, after the first time MEM[A4,M6] modifies the A4 register with M6, the A4 regiter will have the value 0x80000005, the result of 0x8000000 + 5. After another modification using MEM[A4,M6], A4 will not have the value 0x8000000A because this is beyond the range of the L4 register. Instead it will have the result 0x80000002, which is the result of (0x80000005 + 5) - 0x00000008.

The modify values can be used in either pre-modify or post-modify modes. If memory is accessed using MEM[A4,M6], the address output from the address generator will be the value of A4 before the current instruction is executed, and A4 will be adjusted after the memory fetch. Conversely, if memory is accesses using MEM[M6,A4], the address output from the address generator will be the value of the A4 register after being modified by the M6 register.

Figure 6.1 shows the registers and data flow through the address generator. The output from the address register selec-

[1]Because of the design of the address generator, the length register must be a value of 2^n. Other values will have unpredictable results, and should not be used.

tor is added the the output from the modify register selector. This value represents the address after being modified.

The output from the length register is used to generate a pair of bit mask values. These values are used to form the original base address and the modified offset values, limited to within the boundaries of the length register. The value in the Length register must be a valuein the form 2^n to work properly. The first output from the bitmask generator provides the low bit mask, which is logical-ANDed with the sum that was calculated from the selected address and modify registers. The second output from the bit mask generator sets the high bits which is always logical-ANDed with the output of the unmodified address register. This will always result in the same base address of the area. The new address is formed by combining the high bits and low bits using a logical OR. This value can then be loaded bac into the selected Address register.

The address that is output by the address generator may be either the original address stored in the selected address register, or the calculated value that is the result of the address register, modify register, and length registers. If the desired output of the address generator is the unmodified value, this is refered to as a "post-fixed" value; while if the desired output of the address register is the newly calculated value, this is refered to as a "pre-fixed" value. Selection of the "post-fixed" value or the "pre-fixed" value is made by a single control bit coming into the address generator.

Values may be loaded into any of the registers except for M0, M1, and M7 registers. (M0 is pre-wired to always provide a 0, M1 is pre-wired to always provide a +1 value, and M7 is prewired to always provde a -1 value.) A value is loaded into an address register by placing the value on the "Y" bus, selecting which register to place the value in with the 3-bit value, Address select, and finally, a negative pulse on the "/LD_A_reg" signal. The same process is used to load a value into the length registers, except that the "/LD_L_reg" signal will receive the short negative pulse. (The same 3 bits on "Address select" always control both the address and length registers' selections together.) Loading a value into the modify registers (M2-M6)[2] is done by placing the value to load in the "Y" bus, the 3 "Modify select" bits select which register, and a short negative pulse in the "/LD_M_reg" signal.

The address generator can be used to load or store data in memory. If data is to be read from memory, the instruction:

```
R2 = MEM[A3,M5] ;Read from memory, post-fix address
```

will use the address pointed to by the `A3` register, read the data stored in this address, and use the `M5` register to adjust the `A3` register.

Writing data to memory would appear similar. The instruction:

```
MEM[M3,A2] = R18 ;Write to memory, pre-fix address
```

will adjust the `A2` register with `M3`, and the value in `R18` will be stored in memory at the new address location.

Address registers may be adjusted without performing memory accesses. The modify command will perform the same process as a memory access, without doing the memory access, itself. For example, the instruction:

```
MODIFY[A4,M1] ;Increment pointer
```

will automatically increase the `A4` address register without performing a memory access.

The address generator is designed to simplify access to memory. The following excerpt from a 'C' program for example:

```
int buffer[32],i,offset=0;
```

[2] If register M0, M1,or M7 is selected, the value is ignored.

```
for( i=0; i<32; i++ )
{
  buffer[ offset ] = i;
  offset += 5; //interleave array elements by 5;
  if( offset > 32 )
    offset -= 32;
}
```

uses an if-statement to reset back into the buffer's legal region if it exceeds this range. The short segment of code below shows the simple implementation ising the address generator:

```
        ;set up registers
        A0 = (base address of buffer, must have 0 in low 5 bits!)
        L0 = 32  ;Length of the buffer is 32 words
        M2 = 5   ;This is the interleave value
        R1 = 0   ;Initialiaze i in R1 by clearing it to 0

loop:        MEM[A0,M2] = R1        ;Save current value of i, adjust pointe
        R1++             ;Increment i
        CHECK R1 - 32    ;Check if i < 32
        IF LT JUMP loop ;Loop until done (See next chapter)
```

The instruction `MEM[A0,M2] = R1` saves the value in memory and performs the boundary checking automatically.

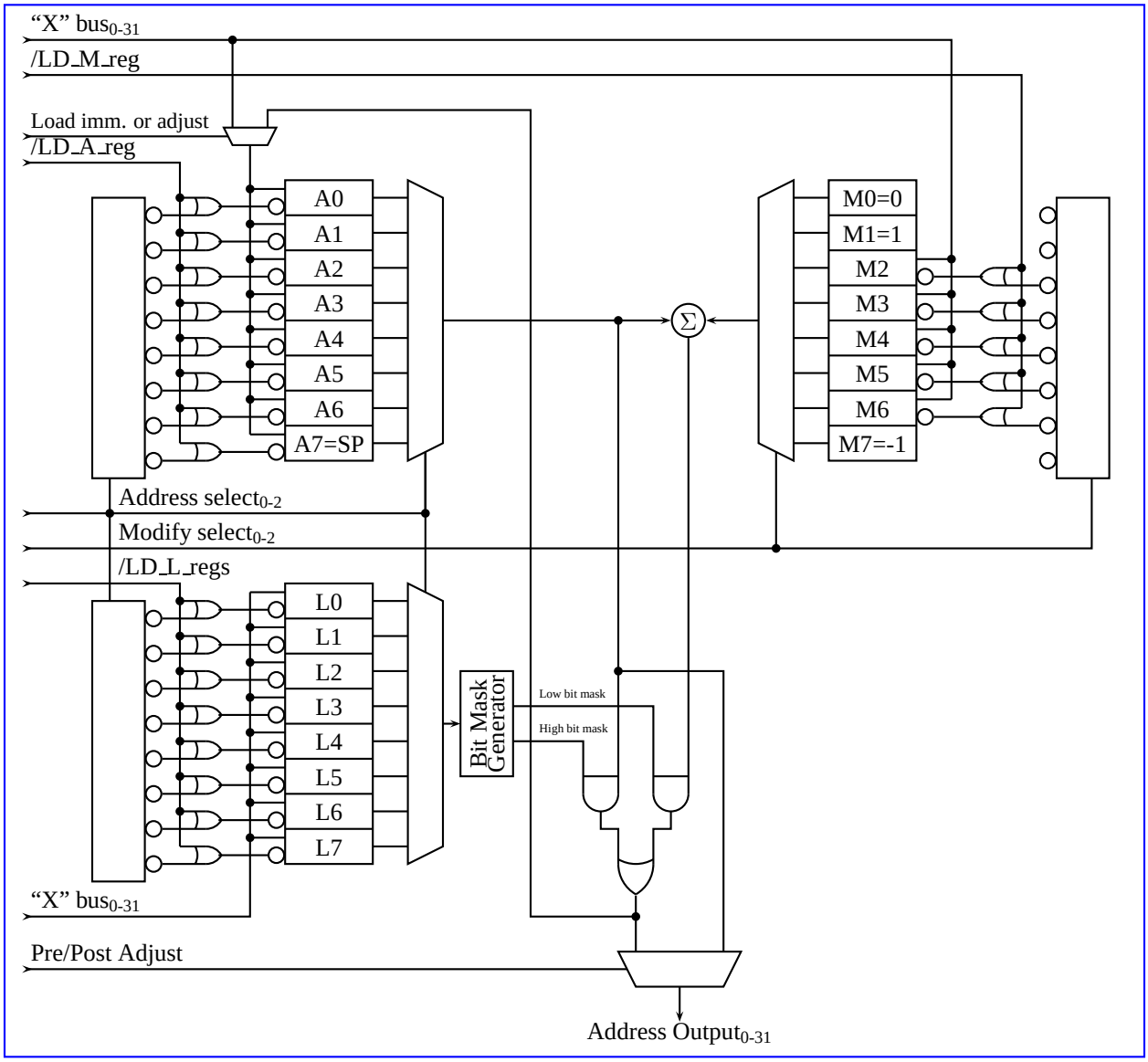


Figure 6.1: Address Generator

Chapter 7

Process Flow Control

There are many ways of effecting the flow of a program being run on the Pioneer-2 processor. This chapter will outline how programs can control the flow of process control such as jumps, branches, and subroutine calls. Other means such as interrupts are described in other chapters.

7.1 Process Flow Control

The most basic component of process flow control is the jump instruction. This allows the processor's PC value to be adjusted to a new value, "jumping" to a different location in the program. In the Pioneer-2 processor, it is not possible to include a 32-bit address value to jump to an immediate location in memory. However, 23 bits are available in the JUMP instruction, and these bits are used to form a signed value, relative to the current location[1]. This jump format is refered to as "JUMP PC-relative" because the destination address is found by adding an offset to the current value of the PC.

Alternatively, one of the general purpose registers may hold an address value, which may indicate the address that should be loaded into the program counter. This format is refered to as a "JUMP Register" because the destination address is the value stored in the selected register.

7.2 Flags

There are four flags available in the Pioneer-2. These are set as results in the various computational units, and can be tested to control instruction flow after a calculation.

The Carry Flag is used by unsigned numbers in the ALU and by the Shifter unit for calculations. In the ALU, it is used to indicate that the result of an addition was over 0x80000000 or that the result the result of a subtraction was below 0. In the shifter, it is used to indicate the last bit that was "lost" as the result of a SHIFT, or it has one of two meanings in a ROTATE instruction. If the instructions is a "rotate-through-carry" (ROTATEC) instruction, the Carry Flag acts as a 33rd bit for the value being rotated. If the ROTATE instruction is not a "rotate-through-carry" instruction, the Carry flag is used to indicate the last bit that was rotated from one end of the register value to the other. (See 5.3 for information on the Carry flag in the SHIFT and ROTATE instructions.) The Carry clag is not effected by the Multiplier/Divider.

[1] The PC always points to the next instruction to be fecthed, so a JUMP +0 will be treated as a NOP instruction.

The Negative Flag is used by signed instructions in the ALU and by the Multiplier/Divider to indicate that the result of a calculation is negative. The Negative Flag is not effected by the Shifter unit.

The oVerflow Flag is used to indicate that the result of a calculation has overflown the data space available to hold the result. This is normally used for signed numbers to indicate that the result of an addition has resulted in a number grater than MAX_POS (`0x7FFFFFFF`) or a subtraction has resulted in a number less than MAX_NEG (`0x80000000`). In the Multoplier, it indicates that the result of an MUL instruction or MULA instruction has exceeded the 32-bit register, and the high 32-bit register has extended bits. Divide instructions will not effect the oVerflow Flag, and the Shifter unit will not effect the oVerflow Flag.

The Zero Flag is used to indicate that the result of a calculation is zero. It is the only flag that is potentially effected by all of the calculations.

The flags are all updated at the end of execution of each instruction.

There are two instructions used to modify the Carry Flag. The `STC` is used to force the flag to a set state, while the `CLC` instruction is used to force the flag to a cleared state.

7.3 Conditional Processing

Every instruction may optionally be prefixed by a conditional statement[2].

For every instruction, the condition codes are checked, and the execution of the instruction is only performed if the condition is satisfied. For example, an instruction prefixed with `IF LT` is only executed if the N (negative) flag is set and the Z (zero) flaf is cleared when the instruction is to be executed. If the N flag is not set, the instruction is ignored.

There are 12 (16?) possible condition codes, read from instruction$_{28-31}$ bits, as seen in 7.1.

Instr$_{31}$	Instr$_{30}$	Instr$_{29}$	Instr$_{28}$	IF condition
0	0	0	0	NEVER (No-Op)
0	0	0	1	(undefined)
0	0	1	0	C (Carry flag set)
0	0	1	1	NC (Carry flag clear)
0	1	0	0	GT (Negative flag clear AND Zero flag clear)
0	1	0	1	GE (Negative flag clear)
0	1	1	0	LT (Negative flag set AND Zero flag clear)
0	1	1	1	LE (Negative flag set OR Zero flag set)
1	0	0	0	NV (Overflow flag clear)
1	0	0	1	V (Overflow flag set)
1	0	1	0	NZ (Zero flag clear)
1	0	1	1	Z (Zero flag set)
1	1	0	0	(undefined)
1	1	0	1	(undefined)
1	1	1	0	(undefined)
1	1	1	1	ALWAYS (default)

Table 7.1: Conditional Processing Codes

[2]Every instruction is prefixed; by default, the assumed condition is “IF ALWAYS” to allow unchanged instructions to assume to be performed.

7.4 Branches

Because all instructions are prefixed by a conditional control, any JUMP instruction may be a branch. The assumed condition for all instructions is IF ALWAYS so that instructions will always be executed by default. Therefore, a JUMP instruction may be prdicated by IF LT to become IF LT JUMP +800 which is a conditional jump, or a "branch."

The JUMP instruction includes a signed offset value as it's argument. If a program wishes to jump to an absolute value, the assembler must manage calculating the offset value to create the lower 23 bits of the address, so the PC may be modified by -2^{22} through $2^{22}-1$ or -4194304 t0 +4194303. This value is always relative to the address stored in the PC, so a jump to relative addres +0 would always execute the next instruction in memory.

7.5 Subroutine Utility Functions

7.5.1 JAL (CALL)

There are many functions designed for support subroutines. The first of these functions is the CALL function.

In the preliminary version of the Pioneer-2, the CALL instruction copies the value from the PC register (with one added to it to correct for the address of the instructino to return to), into register R31. The Pioneer-2 processor decrements the Stack Pointer and stores the return address at this location on the stack[3]. Like the JUMP instructions, the CALL instructions have two modes of operation, both the PC-relative and Register modes can be used.

(CALL system) to be implemented when supervisory/user modes are supported by the processor.

7.5.2 Stack Frame Functions

When a process enters a subroutine, it is typical to create a stack frame, saving critical registers, and creating variable space belonging to the particular instance of a subroutine call. This variable space must be cleared[4] before exiting from the subroutine, and the critical registers are restored before returning to the calling location. The ENTER and LEAVE functions are used to perform this task.

Although these two functions do not save any of the registers, they are used to build a stack frame. Each function receives a parameter that defines how many words to reserve on the stack. The ENTER function subtracts this value from the Stack Pointer, adjusting it to a point that would leave variable space for the subroutine. The LEAVE function adds the specified value to the Stack Pointer, preparing to return from the subroutine.

7.5.3 Stack and Register Functions

Two functions are available to save/restore register values to/from the stack. The PUSH and POP functions are used to push a register onto the stack, or to pop a register value from the stack.

[3] This is a contradiction to most RISC processors which store the return address in one of the general purpose registers.

[4] The stack frame values do not need rewritten, but the Spatck pointer gets adjusted to create the frame, and must be readjusted before exiting from the subroutine.

The PUSH function begins by decrementing the Stack Pointer. The new address in the Stack Pointer is then used to store a selected register. This allows a register's value to be preserved while the register is used for a calculation. Alternatively, it can be used to put a value onto the stack that may be needed by a subroutine.

The POP function reads a value from memory pointed to by the Stack Pointer, then increments the Stack Pointer.

7.5.4 Return from Subroutine

The last subroutine utility function is the RTS instruction. This instruction returns control from the subroutine back to the previous location, where the subroutine was called from. The return address is recalled from the stack, first. Then, the Stack Pointer is incremented to restore it to the same position it was in when the subroutine was called.

In the preliminary version of the Pioneer-2, the RTS has been replaced with a JR instruction, which is used to copy the value of any register into the PC. In the final version, this functionality will be replaced with the MOV instruction, which can still be used to copy any general purpose register to any special purpose register, including the PC. In addition, the JR instruction will go away to make room for the RTS instruction.

RTI (Return from Interrupt/Exception)

Chapter 8

Data Space

Many 32-bit CPUs include addressing which allows the ability to address 8-bit values (bytes) as individual addresses, even though they are subsets of the full 32-bits at each physical address location. Among other complications, these systems suffer huge performance impacts by having misaligned data by havinf values that are split over two seperate physical memory locations.

The Pioneer-2 addresses this issue by not allowing byte-addressed memory space. A memory location is not an 8-bit memory size, but a 32-bit value. As a result, the performance of the Pioneer-2 is improved because the internal hardware is simplified and the word-alignment issues are resolved. If a segment of memory must be compressed into 8-bit sized elements, this must be handled in the software.

Version 1 of the Pioneer-2 uses seperate segments of memory for program memory space and data memory space. The Pioneer-2 is not able to read program instructions from the data memory space, and it is not able to read or write data into the program memory space.

The Address Generator and it's instruction set are used to generate addresses pointing into the data memory. Another addressing mode uses the general purpose registers to forma a base address, and an 11-bit signed value is added the the address stored in the general purpose register. This address can be used to store data or load data from memory.

```
R2 = MEM[ R12 ],45  ;Use the data in R12 as an address, adding 45
                    ;to that, reading that memory address location,
                    ;and store that value in R2
MEM[ R5 ],-2 = R6   ;Use the data in R5 as an address, subtract 2,
                    ;and use that as the address to store register
                    ;R6 into that location
```

Data Memory can be addressed using many different modes from several different registers. This allows maximum flexability and takes advantage of technologies from several different families of CPUs.

Chapter 9

Instruction Set Encoding

Ths instruction set of the Pioneer-2 is designed to fit all instructions into a single 32-bit word. All of the instructions can be fetched in a single memory read access.

Most of the information in different instructions is stored in the same group of fields within the instruction. The high 4 bits of all instructions carry the conditional information, and the nest highest 2 bits always identify which group, ALU, Shift/Multiply/Divide, Subrtoutine/Register, or Special Storage. The generic format of most instructions is seen in Figure 9.1.

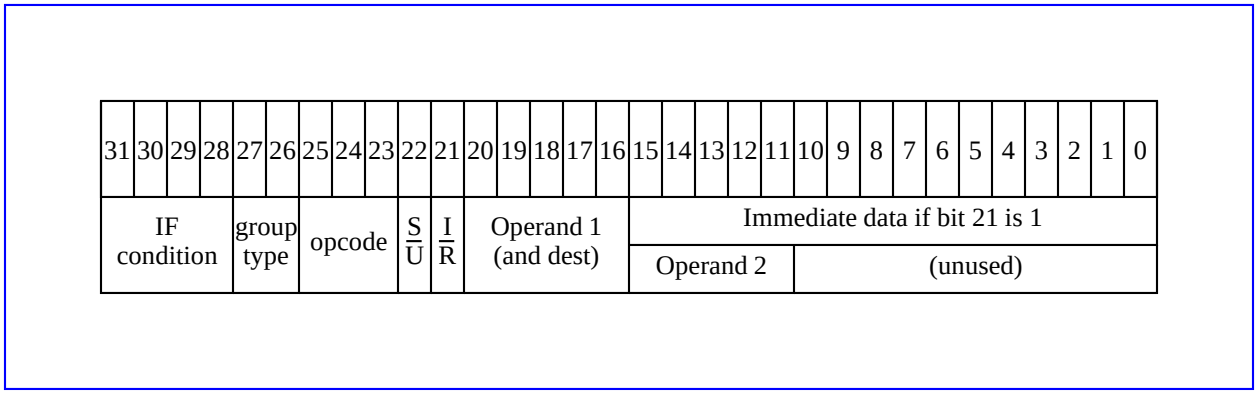


Figure 9.1: Generic Instruction Format

The generic format in Figure 9.1 shows the fields that are most common in most of the instructions, including the destination register (which is also the first operand), as well as the second operand, if required. There are two bits in the instruction field that are used to modify the behavior of most instructions. Instruction$_{22}$ is typically used to indicate a signed instruction if set, or an unsigned instruction if cleared. Instrustion$_{21}$ is typically used to indicate that the instruction's second operand is another register if cleared, or if it is 16 bits of immediate data, read from the low 16 bits of the instruction, if the bit is set.

Instruction$_{26,27}$ bits identify whether the istruction belongs to the ALU, Shifter or Multiplier/Divider Units, Subroutine and Register Operations, or the Special Storage groups of instructions. Instruction$_{23-25}$ bits, and in some cases, instruction$_{21,22}$ bits, identify which specific instruction is to be performed.

9.1 ALU Instruction Set

The ALU instructions are selected when instruction$_{26,27}$ are both 0. The 3 bits, instruction$_{23-25}$ indicate the operation to perform, according to Table 5.1.

Instruction$_{21}$ indicates whether the second operand of this instruction is a register selection or immediate data. If the bit is cleared, the second operand of this instruction is a register, selected by instruction$_{11-15}$. If this bit is set, the second operand of this instruction is 16 bits of immediate data, which is built from instruction$_{0-15}$.

The first operand is a register, which will also be the destination register. This register is selected by instruction$_{16-20}$.

9.2 Shifter Unit Instruction Set

The Shifter Unit and the Multiplier/Divider Unit is selected when instruction$_{27}$ is 0 and instruction$_{26}$ is 1. Furthermore, the Shifter Unit is selected when instruction$_{25}$ is 0.

Instruction$_{23,24}$ are used to select the operation of the shifter. Instruction$_{23}$ is used to indicate the direction of the shift. A 0 in this bit will indicate a left or upward shift, but a 1 in this bit will indicate a right or downward shift. Instruction$_{24}$ is used to indicate whether this operation is a shift or a rotate operation. All of the shifter operations are listed in Table 5.2 and in Figure 5.3.

The sign bit from instruction$_{22}$ is used differently for shift instructions. It is ignored by the LSHIFT instruction, but it is used to change a RSHIFT instruction to an ASHIFT ("Arithmetic RSHIFT") instruction. The RSHIFT and ASHIFT instructions vary in how they adjust bits from the left. The RSHIFT instruction always fills the highest bit with a 0 as it shifts downward, but the ASHIFT always keeps the same bit value in the highest order bit to preserve whether this number is positive or negative. The LSHIFT always pushes a new value into the highest order bit, so the sign if the result is always dependant on the data being shifted upwards, so the sign bit of the instruction has no meaning in this case.

The sign bit from instruction$_{22}$ is also used to modify the ROTATE instructions. When the sign bit is cleared, ROTATE instructions rotate data in a circular motion, and the value of the last bit that "jumps" from the LSB to the MSB (or from the MSB to the LSB, depending on the direction) is always placed in the Carry flag. However, if the sign bit from the instruction is set, the Carry flag is always treated as a 33rd bit to be included in the rotate operation.

Shifter operations require two operands. The first operand is the register that stores the data to be shifted. It is both the source and the destination register, and is selected by instruction$_{16-20}$. The second operand identifies the distance the value is to be shifted. If instruction$_{21}$ is set, the second operand is the immediate data stored in instruction$_{0-15}$, bit if instruction$_{21}$ is cleared, the second operand is a register, selected by instruction$_{11-15}$. In either case, only the bottom 5 bits of the second operand are used in the shoft operation.

9.3 Multiplier/Divider Instruction Set

The Multiplier/Divider Unit is selected like the Shifter Unit, when instruction$_{27}$ is 0 and instruction$_{26}$ is 1. The instruction$_{25}$ bit must be a 1 for the Multiplier/Divider Unit, to distinguish it's instructions from the Shifter Unit instructions.

Instruction$_{24}$ is used to select multiplication when cleared, and to select division when set. Instruction$_{23}$ is ignored for division, but in multiplication, it is used to enable the accumulator, in order to perform a multiply-and-accumulate

(MULA) instruction.

The sign bit of the instruction, $instruction_{22}$ is used in all multiply and divide instructions to indicate whether both operands of the instrustion are signed or unsigned[1].

The first and second operands of the multiply and divide instrustions are controlled by the same bits and the same modes as the ALU instructions.

9.4 Subroutine and Address Generator Instruction Set

9.4.1 Process Flow Instructions

The JAL (CALL) instruction is used to call a subroutine at a location in memory that is determined by adding a 23-bit signed value to the current PC. The original value of the PC is incremented (because that is the address of the next instruction to execute after retirning from the subroutine), and stored in R31.

In future versions of the Pioneer-2, the PC's incremented value will be stored on the stack instead of R31 to more easily facilitate multiple subroutine calls.

This instruction will be shown in the assembly language as:

```
CALL [± offset]
```

The JUMP instruction is used to modify the PC. The PC is usually incremented to point to the next instruction, but for a jump instruction, the PC has a 23-bit signed value added to it.[2]

The JUMP instruction will be shown in the assembly language as:

```
JUMP [± offset]
```

The JR (Jump to Register) instruction copies the value from a register into the PC. In future versions of the Pioneer-2, this instruction will disappear. The same functionality will come from the MOV instruction, which will be able to move any general purpose register value to any special purpose register, including the PC. This instruction can then be replaced by the RTS (Return from Subroutine) instruction which will pop the return address off of the stack and place it into the PC.

```
JR R12
```

```
RTS
```

RTI

9.4.2 Address Generator Instructions

LD adg

[1] Some instruction sets use different bits to indicate signedness of each oerand.

[2] In most CPUs, the PC is always incremented, and the offest is added to that value. Since in my design, I have to use an adder to get PC+1 into R31 for the call instruction anyway, maybe I should use that to add 1 plus the offset to the PC?

ST adg

9.4.3 Stack Frame Instructions

ENTER

LEAVE

9.5 Special Storage Instruction Set

9.5.1 Stack Storage

PUSH

POP

9.5.2 Register Moves

GP register moves

SP registers and moves

9.5.3 FPU Instructions

9.5.4 Direct Register Instructions

The direct register instructions, `INC`, `DEC`, `CLEAR`, and `NEG` (one's compliment) are selected when the instruction$_{27-23}$ bits form the binary value `0b11100`. Additionally, instruction$_{22,21}$ bits select which operation is being selected, accordint to table 9.1.

instr$_{22}$	instr$_{21}$	operation
0	0	INC
0	1	DEC
1	0	CLR
1	1	NEG

Table 9.1: Register Direct Commands

Instruction$_{20-16}$ bits are used to select which register is to be modified.

9.5.5 Partial Register Load

Two instructions are included which will load half registers. The Load Upper Immediate, (LUI) instruction, selected when instruction$_{27-22}$ form the binary value `0b111010`, will load the 16-bit value from the lower 16 bits of the instruction into the register selected by instruction$_{20-16}$. The lower 16 bits of the selected register will be cleared. The LUI instruction takes the form:

```
R8H = 67
```

where the 'H' (or 'h') appended to the register number indicates a load into the high halfword.

Similarly, the Load Lower Immediate (LLI) instruction, selected when instruction$_{27-22}$ form the binary value `0b111011`, will load the 16-bit value from the lower 16 bits of the instruction into the register selected by instruction$_{20-16}$. The high 16 bits of the selected register are cleared, and no sign-extension is performed. The LLI instruction takes the form:

```
R5L = 34
```

where the 'L' (or 'l') appended to the register number indicates a load into the low halfwaord.

9.5.6 Data Memory Storage

Data may be stored and retrieved from the data memory segment. Storing a 32-bit word in memory is selected wheninstruction$_{27-21}$ form the binary value `0b1111100`. The register selected by instruction$_{20-16}$ bits will be the register holding the data to be stored. The register selected by instruction$_{15-11}$ will be used as a base address. Instruction$_{10-0}$ form an 11-bit signed offest to be sign-extended to the address stored in the base register. The data is then stored into data memory. The Store Word (SW) instruction takes the form:

```
DM[R4],68 = R2 ;store R2 in data memory pointed to by R4, plus 68

DM[R11],-34 = R9 ;store R9 in data memory pointed to by R11, minus 34
```

Data may also be read from data memory with the instructions:

```
R6 = DM[R3],12 ;load data memory pointed to by R3, plus 12, into R6
```

The instruction set encoding for the Load Word (LW) instruction is exactly the same as the SW instruction, except that instruction$_{23}$ must be a 0 instead of a 1.

9.6 Complete Instruction Set Map

The complete instruction set map is seen in Figure 9.2.

31

30 29 28	27 26	25 24 23	22	21	20 19 18 17 16	15 14 13 12 11	10 9 8 7 6 5 4 3 2 1 0
IF condition 0000:NEVER 0001:undef 0010:C 0011:NC 0100:GT 0101:GE 0110:LT 0111:LE 1000:NV 1001:V 1010:NZ/NE 1011:Z/EQ 1100:undef 1101:undef 1110:undef 1111:ALWAYS	00= ALU	000=AND 001=OR 010=XOR 011=PASS 100=ADD 101=SUB 110=NEG 111=CMP	0=unsigned, 1=signed	0=register, 1=immediate	"X" bus register and destination	"Y" bus register if instruction$_{21}$=0	(unused) if instruction$_{21}$=0
	01= Shifter or Mult or Divide	000=LSH 001=RSH 010=LROT 011=RROT 100=MUL 101=MULA 110=DIV 111=undef				immediate data if instruction$_{21}$=1	
	10= subrt and Add. Gen.	000=CALL 001=JUMP	23-bit signed offset to add to Program Counter				
		010=RTS 011=RTI	0	0	Non-0 values for instruction$_{21,22}$ enter protected-mode operations		
		100=LDag 101=STag	P*	X	A reg select	M reg select	13-bit signed offset to add to address
		110=ENT 111=LVE	0	22-bit unsigned length to adjust stack frame			
	11= special storage	000=CLC 001=STC	0	(unused)			
		000=PUSH 001=POP	1	G*	register	(unused)	
		010=MOV	G*	G*	dest. reg.	src. reg.	(unused)
		011=FLOP	0	0	"X" bus register	"Y" bus register	floating point instruction
		100=INC 100=DEC 100=CLR 100=NEG	0 0 1 1	0 1 0 1	register	(unused)	
		101=LUI 101=LLI	0 1	X	register	immediate data	
		110=LD 111=ST	0	0	data register	address register	signed offset for addess

*P:Modify Address Register; 0=prefix address, 1=postfix address
*G:Group Select; 0=general registers, 1=special registers

Figure 9.2: Complete Instruction Set Map

Chapter 10

CPU Control Module

The Pioneer-2is controlled by the control module. The control module has 6 inputs: the 32-bit instruction word is used to decode operations and determine how to control the other signals. It also inputs the four system flags do control branching. It inputs the system clock to control timing within the processor, and an active low reset signal.

The control module has many outputs to control each of the components in the Pioneer-2. The first three outputs are used for timing, indicating three phases of the system clock, first, half, and tail[1]. Four signals control the PC and it's surrounding logic; PC_load_select determines whether the PC should load from an adder (used for JUMP and CALL instructions) or the output from the register file (used for return instructions), PC_increment is used to increment the PC, PC_load is used to load an external value into the PC, and link_register is used to force the "Y_sel" input to select register 31.

The register file is controlled by six signals from the control module. The _enable, _load, _clear, count, up_down, and ones_compliment signals from the control module control the register file as described in the Registers chapter. An additional signal, "immediate_Y_mode," is used to select whether the "Y" output from the register file or the immediate data from the instruction word. There is also a 3-bit "register_load_source" that selects one of eight different input sources to load into the registers.

Three output signals control the operation of the computational units. "ALU_active," "shifter_active," and "multiplier_active" signals are used to indicate that the current instruction uses one of the three units.

Data memory is controlled from the control module with two[2] control signals. The "data_memory_select" is used to enable the sram module used for data memory, and the "data_memory_write" signal is used to signal a write into data memory. Future versions are going to use a memory controller and cache system.

Systems flags in the CCR are controlled by 4 signals from the cyctem controller. A 2-bit signal, "flag_sel," is used to select whether the flag outputs from the registers, ALU, shifter, or multiplier are to be used to load into the CCR. These flags are loaded with the "flag_load" signal. Two signals, "CLC" and "STC" are used to force the carry flag to clear or set, respectively.

Internally, the control module checks each instruction's conditional flags and the system's CCR to determine whether to do the instruction or not. This determination is controlled by a signal named "do_operation." If the control module detects that a condition code and system flags indicate not to perform a calculation (because the "do_operation" signal

[1] There is actually a fourth phase of the system clock which may be eliminated, but in future versions of the /cpuname, this will be used as a pre-fetch phase for following instructions.

[2] There was a third control signal, "data_memory_buffer" that was eliminated. It is still included in the current verilog modules, but will be removed later.

is low), it will cause the timing module to skip from “start” phase to “tail” phase, skipping the “half” phase, performing calculations. This will increase the system performance by reducing the cycle from 4 phases to 3[3] for instructions that do not perform calculations. This effect can be seen in an excerpt of the test results:

```
time sht  PC        instr                          CNVZ do_operation
   0 000: 00000000 :xxxxxxxx: 0=xxxxxxxx 1=xxxxxxxx 0000 OK=x
 100 100: 00000000 :fc000000: 0=xxxxxxxx 1=xxxxxxxx 0000 OK=1
 110 010: 00000000 :fc000000: 0=xxxxxxxx 1=xxxxxxxx 0000 OK=1
 120 001: 00000000 :fc000000: 0=xxxxxxxx 1=xxxxxxxx 0000 OK=1
 130 000: 00000001 :fc000000: 0=xxxxxxxx 1=xxxxxxxx 0000 OK=1
 140 100: 00000001 :fe80f234: 0=xxxxxxxx 1=xxxxxxxx 0000 OK=1
 150 010: 00000001 :fe80f234: 0=xxxxxxxx 1=xxxxxxxx 0000 OK=1
 160 001: 00000001 :fe80f234: 0=f2340000 1=xxxxxxxx 0000 OK=1
 170 000: 00000002 :fe80f234: 0=f2340000 1=xxxxxxxx 0000 OK=1
 180 100: 00000002 :f1810000: 0=f2340000 1=xxxxxxxx 0000 OK=1
 190 010: 00000002 :f1810000: 0=f2340000 1=xxxxxxxx 0000 OK=1
 200 001: 00000002 :f1810000: 0=f2340000 1=f2340000 0000 OK=1
 210 000: 00000003 :f1810000: 0=f2340000 1=f2340000 0100 OK=1
 220 100: 00000003 :f2000800: 0=f2340000 1=f2340000 0100 OK=1
 230 010: 00000003 :f2000800: 0=f2340000 1=f2340000 0100 OK=1
 240 001: 00000003 :f2000800: 0=e4680000 1=f2340000 0100 OK=1
 250 000: 00000004 :f2000800: 0=e4680000 1=f2340000 1100 OK=1
 260 100: 00000004 :f2000800: 0=e4680000 1=f2340000 1100 OK=1
 270 010: 00000004 :f2000800: 0=e4680000 1=f2340000 1100 OK=1
 280 001: 00000004 :f2000800: 0=d69c0001 1=f2340000 1100 OK=1
 290 000: 00000005 :f2000800: 0=d69c0001 1=f2340000 1100 OK=1
 300 100: 00000005 :22000800: 0=d69c0001 1=f2340000 1100 OK=1
 310 010: 00000005 :22000800: 0=d69c0001 1=f2340000 1100 OK=1
 320 001: 00000005 :22000800: 0=c8d00002 1=f2340000 1100 OK=1
 330 000: 00000006 :22000800: 0=c8d00002 1=f2340000 1100 OK=1
 340 100: 00000006 :31800800: 0=c8d00002 1=f2340000 1100 OK=0
 350 001: 00000006 :31800800: 0=c8d00002 1=f2340000 1100 OK=0
 360 000: 00000007 :31800800: 0=c8d00002 1=f2340000 1100 OK=0
 370 100: 00000007 :fc000000: 0=c8d00002 1=f2340000 0100 OK=1
 380 010: 00000007 :fc000000: 0=c8d00002 1=f2340000 0100 OK=1
 390 001: 00000007 :fc000000: 0=c8d00002 1=f2340000 0100 OK=1
 400 000: 00000008 :fc000000: 0=c8d00002 1=f2340000 0100 OK=1
```

The excerpt shows each instruction has 4 cycles, shown in the “sht” column, “100” indicates being in the “start” phase, “010” indicates the “half” phase, “001” indicates the “tail” phase, and “000” indicates phase 4, where memory is prefetched. The time column originally shows a 100ns reset before the first instruction (at memory address 00000000) is executed. Note that in phase 4, the program counter (PC) has already been incremented to the next address. The “CNVZ” column indicates the status of the Carry, Negative, oVerflow, and Zero flags, in that order.

At time $\bar{3}00$, the CPU finds the instruction `22000800` which is the instruction, `IF C R0=R0+R1`, and the carry flag is set. The “do_operation” flag is set, indicateing that the instruction will execute the operation. At time $\bar{3}10$, the

[3] It may be possible to reduce this to 2 if a bug can be worked out that keeps the controller from jumping to “phase4” instead of just to “tail” phase. In this case, I think that the PC_increment needs to be sent as soon as do_operation is detected, then the timing controller can jump to phase 4 instead of phase 3.

processor enters phase 2, where R0 and R1 are added, then when the processor enters phase 3, R0 is loaded with the new value. This shows the processor performing a conditional calculation where the instruction is executed.

Conversely, at time $\bar{3}40$, the CPU finds the instruction `31800800` which is the instruction, `IF NC R0 = PASS R1`, and the carry flag is still set. Because the conditional phrase is "IF NC" ("if no carry"), the instruction should not be executed. The "do_operation" flag is cleared, and at time $\bar{3}50$, the CPU skips over phase 2, and R0 is not modified. This shows the processor performing a conditional calculation where the instruction is not executed.

The verilog source code is found in the file, `controller.v`:

```
module if_OK( good, i31, i30, i29, i28, c, n, v, z );
  output good;
  input  i31, i30, i29, i28; //instruction word bits
  input  c, n, v, z;         //flags

  wire if_c;
  wire if_nc;
  wire if_gt;
  wire if_ge;
  wire if_lt;
  wire if_le1;
  wire if_le2;
  wire if_nv;
  wire if_v;
  wire if_nz;
  wire if_z;
  wire if_a;

  and ( if_c, ~i31, ~i30, i29, ~i28, c );
  and ( if_nc, ~i31, ~i30, i29, i28, ~c );
  and ( if_gt, ~i31, i30, ~i29, ~i28, ~z, ~n );
  and ( if_ge, ~i31, i30, ~i29, i28, ~n );
  and ( if_lt, ~i31, i30, i29, ~i28, n, ~z );
  and ( if_le1, ~i31, i30, i29, i28, n );
  and ( if_le2, ~i31, i30, i29, i28, z );
  and ( if_nv, i31, ~i30, ~i29, ~i28, ~v );
  and ( if_v, i31, ~i30, ~i29, i28, v );
  and ( if_nz, i31, ~i30, i29, ~i28, ~z );
  and ( if_z, i31, ~i30, i29, i28, z );
  and ( if_a, i31, i30, i29, i28 );

  or ( good, if_c, if_nc, if_gt, if_ge, if_lt, if_le1,
             if_le2, if_nv, if_v, if_nz, if_z, if_a );
endmodule

module timing( start, half, tail, external_clock, _reset );
  output start, half, tail;
  input  external_clock, _reset;

  reg [1:0] phase;
  reg start, half, tail;
```

```
  always @( negedge external_clock )
    begin
      if (~_reset)
        begin
          phase=3;
          start=0;//1
          half=0;
          tail=0;
        end
      else
        begin
          phase = ( phase + 1 );// & 2'h3;
          case( phase )
            0 : begin
                  start=1; half=0; tail=0;
                end
            1 : begin
                  start=0; half=1; tail=0;
                end
            2 : begin
                  start=0; half=0; tail=1;
                end
            3 : begin
                  start=0; half=0; tail=0;
                end
          endcase
        end
    end
endmodule

module controller( start, half, tail,
                   PC_load_select, PC_increment, PC_load, link_register,
                   _enable, _load, _clear, count, up_down, ones_comp,
                   immediate_Y_mode, register_load_source,
                   ALU_active, shifter_active, multiplier_active,
                   data_memory_select, data_memory_write, data_memory_buffer,
                   CLC, STC, flag_sel, flag_load,
                   i, carry, negative, overflow, zero,
                   system_clock, _reset );
  output start, half, tail;
  output PC_load_select, PC_increment, PC_load, link_register;
  output _enable, _load, _clear, count, up_down, ones_comp;
  output immediate_Y_mode;
  output [2:0] register_load_source;
  output ALU_active, shifter_active, multiplier_active;
  output data_memory_select, data_memory_write, data_memory_buffer;
  output CLC, STC;
  output [1:0] flag_sel;
  output flag_load;
  input  [31:0] i; //instruction word
  input  carry, negative, overflow, zero; //flags
  input  system_clock; //clock
  input  _reset; //active low system reset
```

```
    wire do_operation; //a 1 indicates that this instruction should be executed
    //wire P1, P2;

    wire PC_load_1, PC_load_2, PC_load_signal, PC_load_not;
    wire load_1, load_2, load_3, load_4, load_5, load_6;
    wire clear_signal;
    wire count_1, count_2, count_signal;

    //timing needed to control latches and such
    timing sequencer( start, half, tail, system_clock, _reset );

    //include an IDLE in here later
    if_OK check_flags ( do_operation, i[31], i[30], i[29], i[28],
                                carry, negative, overflow, zero );

    //build the PC_load_select
    and ( PC_load_select, ~i[24], ~i[25], ~i[26], i[27] );

    //build the PC_load (depend on do_operation)
    and ( PC_load_1,                 ~i[24], ~i[25], ~i[26], i[27] ); //true if this is a
    and ( PC_load_2, ~i[21], ~i[22],  i[24], ~i[25], ~i[26], i[27] );
    or  ( PC_load_signal, PC_load_1, PC_load_2 );
    and ( PC_load, do_operation, PC_load_signal, tail ); //only during phase3

    //build the PC_increment (depend on do_operation)
    and ( failedjump, PC_load_1, ~do_operation ); //true when a JUMP statement should not
    xor ( PC_load_not, PC_load_signal, ~failedjump ); //flip load signal if failed jump
    and ( PC_increment, PC_load_not, _reset, tail ); //only during phase3, only while no r

    //build the link_register
    and ( link_register, ~i[23], ~i[24], ~i[25], ~i[26], i[27] );

    //build the _enable (active_low) (depend on do_operation)
    and ( not_enable, i[23], ~i[24], ~i[25], ~i[26], i[27] );
    or  ( _enable, ~do_operation, not_enable, ~_reset ); //force enable high if reset is l

    //build the _load (active_low)
    and ( load_1,  i[23],  i[24],  i[25] ); //any ccccxx111...
    or  ( load2a,  i[23],  i[24],  i[25] ); //look for any non-0
    and ( load_2, load2a, ~i[26],  i[27] ); //any cccc10(non-000)...
    and ( load_3,         ~i[23], ~i[24],  i[25],  i[26],  i[27] );
    and ( load_4,         ~i[23], ~i[24], ~i[25],  i[26],  i[27] );
    and ( load_5, ~i[22],  i[23], ~i[24], ~i[25],  i[26],  i[27] ); //cccc110010...
    and ( load_6,  i[22], ~i[23],  i[24], ~i[25],  i[26],  i[27] ); //cccc110101...

    or  ( _load, load_1, load_2, load_3, load_4, load_5, load_6, ~tail ); //only low durin

    //build the _clear (active_low)
    and ( clear_signal, ~i[21], i[22], ~i[23], ~i[24], i[25], i[26], i[27], half ); //only
    not ( _clear, clear_signal );

    //build the count in phase2
    and ( count_1,  i[21],  i[22], ~i[23], ~i[24], i[25], i[26], i[27] ); //detect NOT Rx
    and ( count_2,         ~i[22], ~i[23], ~i[24], i[25], i[26], i[27] ); //detect INC/DEC
```

```
    or  ( count_signal, count_1, count_2 ); //either NOT or INC/DEC
    and ( count, half, count_signal );

    //build the up_down
    and ( up_down, ~i[21], ~i[22] );

    //build the ones_comp
    and ( ones_comp, i[21], i[22] );

    //build the immediate_Y_mode
    buf ( immediate_Y_mode, i[21] );

    //build the ALU_active
    and ( ALU_active, do_operation, ~i[26], ~i[27] );

    //build the shifter_active
    and ( shifter_active, do_operation, ~i[25], i[26], ~i[27] );

    //build the multiplier_active
    and ( multiplier_active, do_operation, i[25], i[26], ~i[27] );

    //build the data_memory_select
    and ( data_memory_select, do_operation, ~i[21], ~i[22], i[24], i[25], i[26], i[27] );

    //build the data_memory_write
    and ( data_memory_write, i[23], half );

    //build the data_memory_buffer
    buf ( data_memory_buffer, i[23] );

    //build the register_load_source
    and ( div_only, multiplier_active, i[24] );
    and ( lui_instruction, i[23], ~i[24], i[25], i[26], i[27] );
    or ( register_load_source[2], ALU_active, shifter_active, data_memory_select, div_only
    or ( register_load_source[1], multiplier_active, data_memory_select );
    or ( register_load_source[0], shifter_active, data_memory_select, lui_instruction );

    //build flag_sel
    //00 = register flags
    //01 = ALU flags
    //10 = shifter flags
    //11 = multiplier flags
    or  ( flag_sel[0], ALU_active, multiplier_active );
    or  ( flag_sel[1], shifter_active, multiplier_active );

    //build flag_load
    and ( reg_flags, ~_enable, count_signal );
    or  ( update_flags, reg_flags, ALU_active, shifter_active, multiplier_active );
    and ( flag_load, update_flags, tail );

    //build CLC
    and ( CLC, do_operation, ~i[22], ~i[23], ~i[24], ~i[25],  i[26],  i[27] );

    //build STC
```

```
  and ( STC, do_operation, ~i[22],  i[23], ~i[24], ~i[25],  i[26],  i[27] );
endmodule
```

Finally, the entire Pioneer-2 is defined in the verilog module, CPU.v:

```
‘include "controller.v"
‘include "sr_ff_w_gate.v"  //for simplereg
‘include "jk_ms_ff.v"      //for register
‘include "simplereg.v"     //for various latches
‘include "selector32.v"    //32-bit muxes
‘include "decoder.v"       //for reg_file
‘include "register.v"      //already included from register_file.v
‘include "reg_file.v"      //for groups of registers
‘include "flag_reg.v"      //for flags
‘include "sram.v"          //for Data Memory and Program Memory
‘include "add32.v"         //for branch and offset addresses
‘include "addsub32.v"      //For ALU
‘include "alu.v"           //the ALU
‘include "shifter.v"       //the shifter
‘include "u_mul_div.v"     //the multiplier/divider

module fivebit_mux ( bus5out, in_a, in_b, sel );
  output [4:0] bus5out;
  input  [4:0] in_a, in_b;
  input        sel;

  reg [4:0] bus5out;

  always @( in_a or in_b or sel )
    begin
      bus5out = (sel==0) ? in_a : in_b;
    end
endmodule

module CPU( external_clock, _reset );
  input external_clock, _reset;

  wire [31:0] x_bus, y_bus, y_reg, ALU_result, shifter_result,
              multiplier_result_h, multiplier_result_l,
              data_memory_data, data_memory_address,
              program_memory_data, program_memory_address,
              instruction_word, immediate_sign_ext, immediate11_sign_ext,
              immediate_LUI, immediate_LLI, immediate_half_load,
              PC_plus_one, PC_offset, PC_plus_offset, PC_bus, PC_load_value,
              register_load_value, PM_nc, PC_add_sign_ext,
              PC_output, IW_output, x_reg_out, ALU_out, shifter_out,
              ALU_x, ALU_y, shifter_x, shifter_y, multiplier_x, multiplier_y;
              //PC_output is latched data
              //IW_output is latched data
              //x_reg_out is unlatched data
```

```
    wire [2:0]  register_load_source;
    wire [4:0]  x_sel, y_sel;
    wire [1:0]  flag_sel; //which unit provides flag update
    wire        PC_load_select;
    wire        PC_increment;
    wire        PC_load;
    wire        link_register;
    wire        _enable, _load, _clear, count, up_down, ones_comp; //regs
    wire        immediate_Y_mode;
    wire        ALU_active;
    wire        shifter_active;
    wire        multiplier_active;
    wire        data_memory_select, data_memory_write, data_memory_buffer;

    //lots of individual signals to carry flag information around
    wire        register_carry_set, alu_carry_set, shifter_carry_set;
    wire        register_negative_set, alu_negative_set,
                shifter_negative_set, multiplier_negative_set;
    wire        register_overflow_set, alu_overflow_set,
                shifter_overflow_set, multiplier_overflow_set;
    wire        register_zero_set, alu_zero_set,
                shifter_zero_set, multiplier_zero_set;
    wire        CLC; //1=clear carry flag
    wire        STC; //1=set carry flag
    wire        flag_load; //1=load flag register (CCR)
    wire        carry_flag, negative_flag, overflow_flag, zero_flag;
    wire        carry_set, negative_set, overflow_set, zero_set;
    wire        start, half, tail;

    controller cntrl( start, half, tail,
                      PC_load_select, PC_increment, PC_load, link_register,
                      _enable, _load, _clear, count, up_down, ones_comp,
                      immediate_Y_mode, register_load_source,
                      ALU_active, shifter_active, multiplier_active,
                      data_memory_select, data_memory_write, data_memory_buffer,
                      CLC, STC, flag_sel, flag_load, instruction_word,
                      carry_flag, negative_flag, overflow_flag, zero_flag,
                      external_clock, _reset );

    //build all of the PC counter and surrounding logic
    not (PC__load, PC_load );

    register PC ( PC_bus, PC_C_nc, PC_N_nc, PC_V_nc, PC_Z_nc, //flags not used
                  PC_load_value, _reset, PC__load, 1'b1, PC_increment, 1'b0, 1'b0 );
    select_one_of_two32 PC_load_mux ( PC_load_value, x_bus, PC_plus_offset, PC_load_select

    //sign-extend bit 22+
    buf ( PC_add_sign_ext[31], instruction_word[22] );
    buf ( PC_add_sign_ext[30], instruction_word[22] );
    buf ( PC_add_sign_ext[29], instruction_word[22] );
    buf ( PC_add_sign_ext[28], instruction_word[22] );
    buf ( PC_add_sign_ext[27], instruction_word[22] );
    buf ( PC_add_sign_ext[26], instruction_word[22] );
    buf ( PC_add_sign_ext[25], instruction_word[22] );
```

```
    buf ( PC_add_sign_ext[24], instruction_word[22] );
    buf ( PC_add_sign_ext[23], instruction_word[22] );
    buf ( PC_add_sign_ext[22], instruction_word[22] );
    buf ( PC_add_sign_ext[21], instruction_word[21] );
    buf ( PC_add_sign_ext[20], instruction_word[20] );
    buf ( PC_add_sign_ext[19], instruction_word[19] );
    buf ( PC_add_sign_ext[18], instruction_word[18] );
    buf ( PC_add_sign_ext[17], instruction_word[17] );
    buf ( PC_add_sign_ext[16], instruction_word[16] );
    buf ( PC_add_sign_ext[15], instruction_word[15] );
    buf ( PC_add_sign_ext[14], instruction_word[14] );
    buf ( PC_add_sign_ext[13], instruction_word[13] );
    buf ( PC_add_sign_ext[12], instruction_word[12] );
    buf ( PC_add_sign_ext[11], instruction_word[11] );
    buf ( PC_add_sign_ext[10], instruction_word[10] );
    buf ( PC_add_sign_ext[ 9], instruction_word[ 9] );
    buf ( PC_add_sign_ext[ 8], instruction_word[ 8] );
    buf ( PC_add_sign_ext[ 7], instruction_word[ 7] );
    buf ( PC_add_sign_ext[ 6], instruction_word[ 6] );
    buf ( PC_add_sign_ext[ 5], instruction_word[ 5] );
    buf ( PC_add_sign_ext[ 4], instruction_word[ 4] );
    buf ( PC_add_sign_ext[ 3], instruction_word[ 3] );
    buf ( PC_add_sign_ext[ 2], instruction_word[ 2] );
    buf ( PC_add_sign_ext[ 1], instruction_word[ 1] );
    buf ( PC_add_sign_ext[ 0], instruction_word[ 0] );

    reg32 PC_hold( PC_output, PC_bus, start );
    reg32 IW_hold( IW_output, instruction_word, start );
    hold32 ALU_hold( ALU_result, ALU_out, half, _reset );
    reg32 Shift_hold( shifter_result, shifter_out, half );

    fast_adder PC_offset_adder ( PC_plus_offset, PC_o_a_nc, PC_output, PC_add_sign_ext, 1'
    fast_adder PC_next_instr ( PC_plus_one, PC_n_i_nc, PC_output, 32'h00000001, 1'b0 );

    //build the register file and surrounding logic
    //y_sel[4:0] = instruction_word[15:11];
    buf ( y_sel[4], instruction_word[15] );
    buf ( y_sel[3], instruction_word[14] );
    buf ( y_sel[2], instruction_word[13] );
    buf ( y_sel[1], instruction_word[12] );
    buf ( y_sel[0], instruction_word[11] );

    assign immediate_LUI [31:0] = {instruction_word[15:0],16'h0000};
    assign immediate_LLI [31:0] = {16'h0000,instruction_word[15:0]};

    select_one_of_two32 immediate_half ( immediate_half_load, immediate_LUI, immediate_LLI
    assign immediate_sign_ext [31:0] = instruction_word[22] ?
      { instruction_word[15], instruction_word[15], instruction_word[15], instruction_word
        instruction_word[15], instruction_word[15], instruction_word[15], instruction_word
        instruction_word[15], instruction_word[15], instruction_word[15], instruction_word
        instruction_word[15], instruction_word[15], instruction_word[15], instruction_word
        instruction_word[15:0] } :
      {16'h0000,instruction_word[15:0]};
```

```
assign immediate11_sign_ext [31:0] =
  { instruction_word[10], instruction_word[10], instruction_word[10], instruction_word
    instruction_word[10], instruction_word[10], instruction_word[10], instruction_word
    instruction_word[10], instruction_word[10], instruction_word[10], instruction_word
    instruction_word[10], instruction_word[10], instruction_word[10], instruction_word
    instruction_word[10], instruction_word[10], instruction_word[10], instruction_word
    instruction_word[10],
    instruction_word[10:0] };

register_file registers ( x_bus, y_reg, register_carry_set, register_negative_set,
                          register_overflow_set, register_zero_set, register_load_valu
                          x_sel, y_sel,
                          up_down, count, _clear, _load, ones_comp, _enable );
fivebit_mux x_select_mux ( x_sel, instruction_word[20:16], 5'b11111, link_register );
select_one_of_eight32 register_load_mux ( register_load_value,
                                          PC_plus_one,
                                          immediate_half_load,
                                          multiplier_result_l,
                                          32'h0, //will be special reg load
                                          //ALU_result,  take out because acting fun
                                          ALU_out,
                                          shifter_result,
                                          multiplier_result_h,
                                          data_memory_data, register_load_source );
select_one_of_two32 Y_bus_mux ( y_bus, y_reg, immediate_sign_ext, immediate_Y_mode );

//build the computational units
reg32 ALU_X_hold ( ALU_x, x_bus, start );
reg32 ALU_Y_hold ( ALU_y, y_bus, start );

alu ALU ( ALU_out, alu_c, alu_n, alu_v, alu_z, ALU_x, ALU_y,
          instruction_word[25:23], instruction_word[22], carry_flag );
and ( alu_carry_set, alu_c, ALU_active );
and ( alu_negative_set, alu_n, ALU_active );
and ( alu_overflow_set, alu_v, ALU_active );
and ( alu_zero_set, alu_z, ALU_active );

reg32 shifter_X_hold ( shifter_x, x_bus, start );
reg32 shifter_Y_hold ( shifter_y, y_bus, start );

shifter shifter_unit ( shifter_out, shifter_x, shifter_y,
                       instruction_word[24:23], instruction_word[22] );
and ( shifter_carry_set, 0, shifter_active );  //need to fix
and ( shifter_negative_set, shifter_result[31], shifter_active );
and ( shifter_overflow_set, 0, shifter-active ); //always 0
and ( shifter_zero_set, 0, shifter_active ); //need to fix

reg32 multiplier_X_hold ( multiplier_x, x_bus, start );
reg32 multiplier_Y_hold ( multiplier_y, y_bus, start );

mul_div multiplier_unit ( multiplier_result_h, multiplier_result_l,
                          multiplier_x, multiplier_y, instruction_word[24:23] );
and ( multpilier_carry_set, 0, multiplier_active ); //always 0
and ( multiplier_negative_set, 0, multiplier_active ); //need to fix
```

```
   and ( multiplier_overflow_set, 0, multiplier_active ); //need to fix
   and ( multiplier_zero_set, 0, multiplier_active ); //need to fix

   flag_mux select_flags ( carry_set, negative_set, overflow_set, zero_set,
                           register_carry_set, register_negative_set, register_overflow_s
                           alu_carry_set, alu_negative_set, alu_overflow_set, alu_zero_se
                           shifter_carry_set, shifter_negative_set, shifter_overflow_set,
                           multiplier_carry_set, multiplier_negative_set, multiplier_over
                           flag_sel );

   flag_reg flags ( carry_flag, negative_flag, overflow_flag, zero_flag,
                    carry_set, negative_set, overflow_set, zero_set,
                    flag_load, CLC, STC, _reset );

   //memory modules
   sram program_memory ( instruction_word, PM_nc, PC_output[9:0], 1'b0, 1'b1 );
   sram data_memory    ( data_memory_data, x_bus, data_memory_address[9:0], data_memory_w
   fast_adder data_offset_adder ( data_memory_address, DM_o_a_nc, immediate11_sign_ext, y

endmodule
```

Figure 10.1 shows the logical states of the controller module. It shows the 4 phases of most of the instructions, as well as the bypass that is taken when an instruction detects a condition code that does not perform the instruction's operation. In future versions, phase 2 will loop while waiting for long instructions (multiply, divide, long memory fetches) until completion of the calculation.

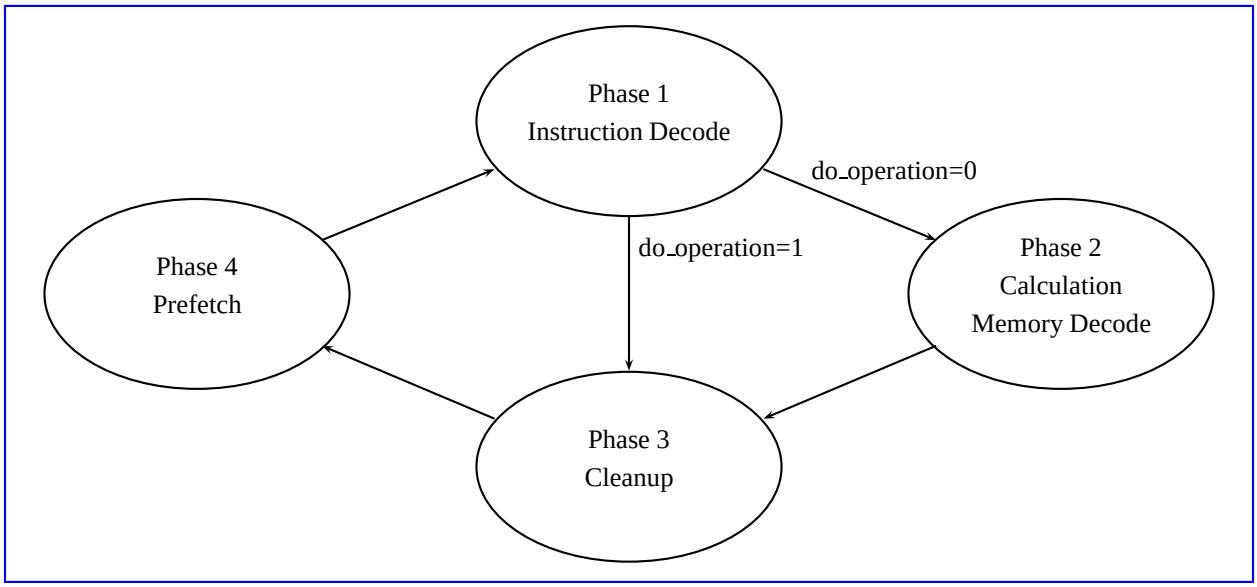


Figure 10.1: Multi-cycle Phase Diagram

Chapter 11

Testing

The Pioneer-2 was tested using a single program that tested the entire set of instructions[1]. To accomplish this, the following program was used.

```
// 0x00000000
FC000000 // IF ALWAYS CLC
FE80F234 // IF ALWAYS R0H=#F234          (proves LUI)
F1810000 // IF ALWAYS R1 = PASS R0       (+ PASS)
F2000800 // IF ALWAYS R0 = R0 + R1       (proves ADDU)
F2000800 // IF ALWAYS R0 = R0 + R1
22000800 // IF C      R0 = R0 + R1       (conditional on carry flag)
31800800 // IF NC     R0 = PASS R1       (conditional on not carry flag)
FC000000 // IF ALWAYS CLC
F2800000 // IF ALWAYS R0 = R0 - R0       (proves SUBU)
BC000000 // IF Z      CLC                (conditional on EQ to 0)
F2800800 // IF ALWAYS R0 = R0 - R1       (another SUBU)
F8000025 // IF ALWAYS CALL 0x00000030    (proves JAL or CALL)
F3000800 // IF ALWAYS R0 = -R1           (+ NEG two's compliment)
F8000023 // IF ALWAYS CALL 0x00000030
F3800800 // IF ALWAYS COMPARE R0 - R1    (CMP SUB with no write, just flags)
00000000 // IF NEVER (NOP)
// 0x00000010
F0000800 // IF ALWAYS R0 = R0 AND R1     (proves AND)
F800001F // IF ALWAYS CALL 0x00000030
F0800800 // IF ALWAYS R0 = R0 OR R1      (proves OR)
F800001D // IF ALWAYS CALL 0x00000030
F1000800 // IF ALWAYS R0 = R0 XOR R1     (proves XOR)
F800001B // IF ALWAYS CALL 0x00000030
F1C00800 // IF ALWAYS R0 = NOT R1        (+ NOT one's compliment)
F8000019 // IF ALWAYS CALL 0x00000030
F2201234 // IF ALWAYS R0 = R0 + #1234    (proves ADDUI)
F8000017 // IF ALWAYS CALL 0x00000030
F2A03F65 // IF ALWAYS R0 = R0 - #3F65    (+ SUBUI)
F8000015 // IF ALWAYS CALL 0x00000030
```

[1] Not all combinations of instructions and condition codes were examined, however. This would be over 600 instructions to execute.

```
F02003C0 // IF ALWAYS R0 = R0 AND #03C0  (proves ANDI)
F0A0F5A3 // IF ALWAYS R0 = R0 OR #F5A3   (proves ORI)
F1203F06 // IF ALWAYS R0 = R0 XOR #3F06  (proves XORI)
B8800000 // IF Z      JUMP * (this instr)(No jump if not =0)
// 0x00000020
F0400800 // IF ALWAYS R0 = R0 NAND R1    (+ NAND)
F800000F // IF ALWAYS CALL 0x00000030
F0C00800 // IF ALWAYS R0 = R0 NOR R1     (+ NOR)
F800000D // IF ALWAYS CALL 0x00000030
F1400800 // IF ALWAYS R0 = R0 XNOR R1    (+ XNOR)
FE400000 // IF ALWAYS CLEAR R0           (+ CLEAR)
FE000000 // IF ALWAYS INC   R0           (+ INC)
FE000000 // IF ALWAYS INC   R0
F2A00002 // IF ALWAYS R0 = R0 - #0002    (R0 should be 0)
A8800003 // IF NZ     JUMP *+3           (prove BNE not branching)
B8800002 // IF Z      JUMP *+2           (prove BEQ branching)
FE80F324 // IF ALWAYS R0H = #F324       (should get skipped)
B8000004 // IF Z      CALL 0x00000030    (conditional CALL)
00000000 // IF NEVER  NOP
00000000 // IF NEVER  NOP
F8800007 // IF ALWAYS JUMP *+7 (0x36)    (proves simple JUMP)
// 0x00000030
FE807654 // IF ALWAYS R0H = #7654
F0A03210 // IF ALWAYS R0 = R0 OR #3210   (proves ORI)
FE811234 // IF ALWAYS R1H = #1234
F0A15678 // IF ALWAYS R1 = R1 OR #5678
F91F0000 // IF ALWAYS JUMP R31 (RTS)     (proves JR31 or RETURN)
00000000 // IF NEVER  NOP                (just space)
F87FFFFA // IF ALWAYS CALL 0x00000030    (Backwards!)
FE400000 // IF ALWAYS CLEAR R0
FE000000 // IF ALWAYS INC R0
F2A00002 // IF ALWAYS R0 = R0 - #0002    (R0 now has -1!)
B8800003 // IF Z      JUMP *+3           (prove BEQ not branching)
A8800002 // IF NZ     JUMP *+2           (prove BNE branching)
F0000800 // IF ALWAYS R0 = R0 AND R1     (should get skipped)
FEE00013 // IF ALWAYS R0L = #13          (+ LLI)
FEE10003 // IF ALWAYS R1L = #3
F4000800 // IF ALWAYS LSHIFT R0 BY R1    (prove SHL)
// 0x00000040
FE210000 // IF ALWAYS DEC R1             (+ DEC)
F4800800 // IF ALWAYS RSHIFT R0 BY R1    (prove SHR)
F4A00001 // IF ALWAYS RSHIFT R0 BY #1    (+ SHRI)
FE80F234 // IF ALWAYS R0H=#F234          (Load high bit)
F4C00800 // IF ALWAYS ASHIFT R0 BY R1    (prove SRA)
F5200003 // IF ALWAYS LROT R0 BY #3      (+ LROTI)
F5A00002 // IF ALWAYS RROT R0 BY #2      (+ RROTI)
F5000800 // IF ALWAYS LROT R0 BY R1      (+ LROT)
FE010000 // IF ALWAYS INC R1
F5800800 // IF ALWAYS RROT R0 BY R1      (+ RROT)
FEE00013 // IF ALWAYS R0L = #13
FEE1000B // IF ALWAYS R1L = #000B
F6000800 // IF ALWAYS R0 = R0 * R1       (prove MUL)
FEE00013 // IF ALWAYS R0L = #13
FEE10003 // IF ALWAYS R1L = #3
```

```
F7000800 // IF ALWAYS R0 = R0 / R1       (prove DIV)
// 0x00000050
FE80F324 // IF ALWAYS R0H = #F324
F0A00ABC // IF ALWAYS R0 = R0 OR #0ABC   (R0=#F2340ABC)
FE410000 // IF ALWAYS CLEAR R1           (R1 points to address 0)
FF800800 // IF ALWAYS DM[R1]0=R0         (prove SW)
FE400000 // IF ALWAYS CLEAR R0           (Clear R0)
FF000800 // IF ALWAYS R0=DM[R1]0         (prove LW)
F8800000 // IF ALWAYS JUMP *             (trap here)

(end of program... remaining memory contents are 0)
```

The output from the test program is shown below. The first column shows the simulation time. The second column shows the phase of each instruction cycle. The PC is shown in the third column, and the current instruction word is shown in the fourth column, between a pair of colons. The contents of registers R0 and R1 are in the fifth and sixth columns, and the last column shows the carry, negative, overflow, and zero flags, concatenated into a 4-bit binary value.

```
$time sht  PC        instr                          CNVZ
    0 000: 00000000 :xxxxxxxx: 0=xxxxxxxx 1=xxxxxxxx 0000
  100 100: 00000000 :fc000000: 0=xxxxxxxx 1=xxxxxxxx 0000
  120 010: 00000000 :fc000000: 0=xxxxxxxx 1=xxxxxxxx 0000
  140 001: 00000000 :fc000000: 0=xxxxxxxx 1=xxxxxxxx 0000
  160 000: 00000001 :fc000000: 0=xxxxxxxx 1=xxxxxxxx 0000
  180 100: 00000001 :fe80f234: 0=xxxxxxxx 1=xxxxxxxx 0000
  200 010: 00000001 :fe80f234: 0=xxxxxxxx 1=xxxxxxxx 0000
  220 001: 00000001 :fe80f234: 0=f2340000 1=xxxxxxxx 0000
  240 000: 00000002 :fe80f234: 0=f2340000 1=xxxxxxxx 0000
  260 100: 00000002 :f1810000: 0=f2340000 1=xxxxxxxx 0000
  280 010: 00000002 :f1810000: 0=f2340000 1=xxxxxxxx 0000
  300 001: 00000002 :f1810000: 0=f2340000 1=f2340000 0000
  320 000: 00000003 :f1810000: 0=f2340000 1=f2340000 0100
  340 100: 00000003 :f2000800: 0=f2340000 1=f2340000 0100
  360 010: 00000003 :f2000800: 0=f2340000 1=f2340000 0100
  380 001: 00000003 :f2000800: 0=e4680000 1=f2340000 0100
  400 000: 00000004 :f2000800: 0=e4680000 1=f2340000 1100
  420 100: 00000004 :f2000800: 0=e4680000 1=f2340000 1100
  440 010: 00000004 :f2000800: 0=e4680000 1=f2340000 1100
  460 001: 00000004 :f2000800: 0=d69c0001 1=f2340000 1100
  480 000: 00000005 :f2000800: 0=d69c0001 1=f2340000 1100
  500 100: 00000005 :22000800: 0=d69c0001 1=f2340000 1100
  520 010: 00000005 :22000800: 0=d69c0001 1=f2340000 1100
  540 001: 00000005 :22000800: 0=c8d00002 1=f2340000 1100
  560 000: 00000006 :22000800: 0=c8d00002 1=f2340000 1100
  580 100: 00000006 :31800800: 0=c8d00002 1=f2340000 1100
  600 010: 00000006 :31800800: 0=c8d00002 1=f2340000 1100
  620 001: 00000006 :31800800: 0=c8d00002 1=f2340000 1100
  640 000: 00000007 :31800800: 0=c8d00002 1=f2340000 1100
  660 100: 00000007 :fc000000: 0=c8d00002 1=f2340000 0100
```

```
  680 010: 00000007 :fc000000: 0=c8d00002 1=f2340000 0100
  700 001: 00000007 :fc000000: 0=c8d00002 1=f2340000 0100
  720 000: 00000008 :fc000000: 0=c8d00002 1=f2340000 0100
  740 100: 00000008 :f2800000: 0=c8d00002 1=f2340000 0100
  760 010: 00000008 :f2800000: 0=c8d00002 1=f2340000 0100
  780 001: 00000008 :f2800000: 0=00000000 1=f2340000 0100
  800 000: 00000009 :f2800000: 0=00000000 1=f2340000 1011
  820 100: 00000009 :bc000000: 0=00000000 1=f2340000 0011
  840 010: 00000009 :bc000000: 0=00000000 1=f2340000 0011
  860 001: 00000009 :bc000000: 0=00000000 1=f2340000 0011
  880 000: 0000000a :bc000000: 0=00000000 1=f2340000 0011
  900 100: 0000000a :f2800800: 0=00000000 1=f2340000 0011
  920 010: 0000000a :f2800800: 0=00000000 1=f2340000 0011
  940 001: 0000000a :f2800800: 0=0dcc0000 1=f2340000 0011
  960 000: 0000000b :f2800800: 0=0dcc0000 1=f2340000 0000
  980 100: 0000000b :f8000025: 0=0dcc0000 1=f2340000 0000
 1000 010: 0000000b :f8000025: 0=0dcc0000 1=f2340000 0000
 1020 001: 00000030 :f8000025: 0=0dcc0000 1=f2340000 0000
 1040 000: 00000030 :f8000025: 0=0dcc0000 1=f2340000 0000
 1060 100: 00000030 :fe807654: 0=0dcc0000 1=f2340000 0000
 1080 010: 00000030 :fe807654: 0=0dcc0000 1=f2340000 0000
 1100 001: 00000030 :fe807654: 0=76540000 1=f2340000 0000
 1120 000: 00000031 :fe807654: 0=76540000 1=f2340000 0000
 1140 100: 00000031 :f0a03210: 0=76540000 1=f2340000 0000
 1160 010: 00000031 :f0a03210: 0=76540000 1=f2340000 0000
 1180 001: 00000031 :f0a03210: 0=76543210 1=f2340000 0000
 1200 000: 00000032 :f0a03210: 0=76543210 1=f2340000 0000
 1220 100: 00000032 :fe811234: 0=76543210 1=f2340000 0000
 1240 010: 00000032 :fe811234: 0=76543210 1=f2340000 0000
 1260 001: 00000032 :fe811234: 0=76543210 1=12340000 0000
 1280 000: 00000033 :fe811234: 0=76543210 1=12340000 0000
 1300 100: 00000033 :f0a15678: 0=76543210 1=12340000 0000
 1320 010: 00000033 :f0a15678: 0=76543210 1=12340000 0000
 1340 001: 00000033 :f0a15678: 0=76543210 1=12345678 0000
 1360 000: 00000034 :f0a15678: 0=76543210 1=12345678 0000
 1380 100: 00000034 :f91f0000: 0=76543210 1=12345678 0000
 1400 010: 00000034 :f91f0000: 0=76543210 1=12345678 0000
 1420 001: 0000000c :f91f0000: 0=76543210 1=12345678 0000
 1440 000: 0000000c :f91f0000: 0=76543210 1=12345678 0000
 1460 100: 0000000c :f3000800: 0=76543210 1=12345678 0000
 1480 010: 0000000c :f3000800: 0=76543210 1=12345678 0000
 1500 001: 0000000c :f3000800: 0=edcba988 1=12345678 0000
 1520 000: 0000000d :f3000800: 0=edcba988 1=12345678 0100
 1540 100: 0000000d :f8000023: 0=edcba988 1=12345678 0100
 1560 010: 0000000d :f8000023: 0=edcba988 1=12345678 0100
 1580 001: 00000030 :f8000023: 0=edcba988 1=12345678 0100
 1600 000: 00000030 :f8000023: 0=edcba988 1=12345678 0100
 1620 100: 00000030 :fe807654: 0=edcba988 1=12345678 0100
 1640 010: 00000030 :fe807654: 0=edcba988 1=12345678 0100
 1660 001: 00000030 :fe807654: 0=76540000 1=12345678 0100
 1680 000: 00000031 :fe807654: 0=76540000 1=12345678 0100
 1700 100: 00000031 :f0a03210: 0=76540000 1=12345678 0100
 1720 010: 00000031 :f0a03210: 0=76540000 1=12345678 0100
 1740 001: 00000031 :f0a03210: 0=76543210 1=12345678 0100
```

```
1760 000: 00000032 :f0a03210: 0=76543210 1=12345678 0000
1780 100: 00000032 :fe811234: 0=76543210 1=12345678 0000
1800 010: 00000032 :fe811234: 0=76543210 1=12345678 0000
1820 001: 00000032 :fe811234: 0=76543210 1=12340000 0000
1840 000: 00000033 :fe811234: 0=76543210 1=12340000 0000
1860 100: 00000033 :f0a15678: 0=76543210 1=12340000 0000
1880 010: 00000033 :f0a15678: 0=76543210 1=12340000 0000
1900 001: 00000033 :f0a15678: 0=76543210 1=12345678 0000
1920 000: 00000034 :f0a15678: 0=76543210 1=12345678 0000
1940 100: 00000034 :f91f0000: 0=76543210 1=12345678 0000
1960 010: 00000034 :f91f0000: 0=76543210 1=12345678 0000
1980 001: 0000000e :f91f0000: 0=76543210 1=12345678 0000
2000 000: 0000000e :f91f0000: 0=76543210 1=12345678 0000
2020 100: 0000000e :f3800800: 0=76543210 1=12345678 0000
2040 010: 0000000e :f3800800: 0=76543210 1=12345678 0000
2060 001: 0000000e :f3800800: 0=76543210 1=12345678 0000
2080 000: 0000000f :f3800800: 0=76543210 1=12345678 1000
2100 100: 0000000f :00000000: 0=76543210 1=12345678 1000
2120 010: 0000000f :00000000: 0=76543210 1=12345678 1000
2140 001: 0000000f :00000000: 0=76543210 1=12345678 1000
2160 000: 00000010 :00000000: 0=76543210 1=12345678 1000
2180 100: 00000010 :f0000800: 0=76543210 1=12345678 1000
2200 010: 00000010 :f0000800: 0=76543210 1=12345678 1000
2220 001: 00000010 :f0000800: 0=12141210 1=12345678 1000
2240 000: 00000011 :f0000800: 0=12141210 1=12345678 0000
2260 100: 00000011 :f800001f: 0=12141210 1=12345678 0000
2280 010: 00000011 :f800001f: 0=12141210 1=12345678 0000
2300 001: 00000030 :f800001f: 0=12141210 1=12345678 0000
2320 000: 00000030 :f800001f: 0=12141210 1=12345678 0000
2340 100: 00000030 :fe807654: 0=12141210 1=12345678 0000
2360 010: 00000030 :fe807654: 0=12141210 1=12345678 0000
2380 001: 00000030 :fe807654: 0=76540000 1=12345678 0000
2400 000: 00000031 :fe807654: 0=76540000 1=12345678 0000
2420 100: 00000031 :f0a03210: 0=76540000 1=12345678 0000
2440 010: 00000031 :f0a03210: 0=76540000 1=12345678 0000
2460 001: 00000031 :f0a03210: 0=76543210 1=12345678 0000
2480 000: 00000032 :f0a03210: 0=76543210 1=12345678 0000
2500 100: 00000032 :fe811234: 0=76543210 1=12345678 0000
2520 010: 00000032 :fe811234: 0=76543210 1=12345678 0000
2540 001: 00000032 :fe811234: 0=76543210 1=12340000 0000
2560 000: 00000033 :fe811234: 0=76543210 1=12340000 0000
2580 100: 00000033 :f0a15678: 0=76543210 1=12340000 0000
2600 010: 00000033 :f0a15678: 0=76543210 1=12340000 0000
2620 001: 00000033 :f0a15678: 0=76543210 1=12345678 0000
2640 000: 00000034 :f0a15678: 0=76543210 1=12345678 0000
2660 100: 00000034 :f91f0000: 0=76543210 1=12345678 0000
2680 010: 00000034 :f91f0000: 0=76543210 1=12345678 0000
2700 001: 00000012 :f91f0000: 0=76543210 1=12345678 0000
2720 000: 00000012 :f91f0000: 0=76543210 1=12345678 0000
2740 100: 00000012 :f0800800: 0=76543210 1=12345678 0000
2760 010: 00000012 :f0800800: 0=76543210 1=12345678 0000
2780 001: 00000012 :f0800800: 0=76747678 1=12345678 0000
2800 000: 00000013 :f0800800: 0=76747678 1=12345678 0000
2820 100: 00000013 :f800001d: 0=76747678 1=12345678 0000
```

```
2840 010: 00000013 :f800001d: 0=76747678 1=12345678 0000
2860 001: 00000030 :f800001d: 0=76747678 1=12345678 0000
2880 000: 00000030 :f800001d: 0=76747678 1=12345678 0000
2900 100: 00000030 :fe807654: 0=76747678 1=12345678 0000
2920 010: 00000030 :fe807654: 0=76747678 1=12345678 0000
2940 001: 00000030 :fe807654: 0=76540000 1=12345678 0000
2960 000: 00000031 :fe807654: 0=76540000 1=12345678 0000
2980 100: 00000031 :f0a03210: 0=76540000 1=12345678 0000
3000 010: 00000031 :f0a03210: 0=76540000 1=12345678 0000
3020 001: 00000031 :f0a03210: 0=76543210 1=12345678 0000
3040 000: 00000032 :f0a03210: 0=76543210 1=12345678 0000
3060 100: 00000032 :fe811234: 0=76543210 1=12345678 0000
3080 010: 00000032 :fe811234: 0=76543210 1=12345678 0000
3100 001: 00000032 :fe811234: 0=76543210 1=12340000 0000
3120 000: 00000033 :fe811234: 0=76543210 1=12340000 0000
3140 100: 00000033 :f0a15678: 0=76543210 1=12340000 0000
3160 010: 00000033 :f0a15678: 0=76543210 1=12340000 0000
3180 001: 00000033 :f0a15678: 0=76543210 1=12345678 0000
3200 000: 00000034 :f0a15678: 0=76543210 1=12345678 0000
3220 100: 00000034 :f91f0000: 0=76543210 1=12345678 0000
3240 010: 00000034 :f91f0000: 0=76543210 1=12345678 0000
3260 001: 00000014 :f91f0000: 0=76543210 1=12345678 0000
3280 000: 00000014 :f91f0000: 0=76543210 1=12345678 0000
3300 100: 00000014 :f1000800: 0=76543210 1=12345678 0000
3320 010: 00000014 :f1000800: 0=76543210 1=12345678 0000
3340 001: 00000014 :f1000800: 0=64606468 1=12345678 0000
3360 000: 00000015 :f1000800: 0=64606468 1=12345678 0000
3380 100: 00000015 :f800001b: 0=64606468 1=12345678 0000
3400 010: 00000015 :f800001b: 0=64606468 1=12345678 0000
3420 001: 00000030 :f800001b: 0=64606468 1=12345678 0000
3440 000: 00000030 :f800001b: 0=64606468 1=12345678 0000
3460 100: 00000030 :fe807654: 0=64606468 1=12345678 0000
3480 010: 00000030 :fe807654: 0=64606468 1=12345678 0000
3500 001: 00000030 :fe807654: 0=76540000 1=12345678 0000
3520 000: 00000031 :fe807654: 0=76540000 1=12345678 0000
3540 100: 00000031 :f0a03210: 0=76540000 1=12345678 0000
3560 010: 00000031 :f0a03210: 0=76540000 1=12345678 0000
3580 001: 00000031 :f0a03210: 0=76543210 1=12345678 0000
3600 000: 00000032 :f0a03210: 0=76543210 1=12345678 0000
3620 100: 00000032 :fe811234: 0=76543210 1=12345678 0000
3640 010: 00000032 :fe811234: 0=76543210 1=12345678 0000
3660 001: 00000032 :fe811234: 0=76543210 1=12340000 0000
3680 000: 00000033 :fe811234: 0=76543210 1=12340000 0000
3700 100: 00000033 :f0a15678: 0=76543210 1=12340000 0000
3720 010: 00000033 :f0a15678: 0=76543210 1=12340000 0000
3740 001: 00000033 :f0a15678: 0=76543210 1=12345678 0000
3760 000: 00000034 :f0a15678: 0=76543210 1=12345678 0000
3780 100: 00000034 :f91f0000: 0=76543210 1=12345678 0000
3800 010: 00000034 :f91f0000: 0=76543210 1=12345678 0000
3820 001: 00000016 :f91f0000: 0=76543210 1=12345678 0000
3840 000: 00000016 :f91f0000: 0=76543210 1=12345678 0000
3860 100: 00000016 :f1c00800: 0=76543210 1=12345678 0000
3880 010: 00000016 :f1c00800: 0=76543210 1=12345678 0000
3900 001: 00000016 :f1c00800: 0=edcba987 1=12345678 0000
```

```
3920 000: 00000017 :f1c00800: 0=edcba987 1=12345678 0100
3940 100: 00000017 :f8000019: 0=edcba987 1=12345678 0100
3960 010: 00000017 :f8000019: 0=edcba987 1=12345678 0100
3980 001: 00000030 :f8000019: 0=edcba987 1=12345678 0100
4000 000: 00000030 :f8000019: 0=edcba987 1=12345678 0100
4020 100: 00000030 :fe807654: 0=edcba987 1=12345678 0100
4040 010: 00000030 :fe807654: 0=edcba987 1=12345678 0100
4060 001: 00000030 :fe807654: 0=76540000 1=12345678 0100
4080 000: 00000031 :fe807654: 0=76540000 1=12345678 0100
4100 100: 00000031 :f0a03210: 0=76540000 1=12345678 0100
4120 010: 00000031 :f0a03210: 0=76540000 1=12345678 0100
4140 001: 00000031 :f0a03210: 0=76543210 1=12345678 0100
4160 000: 00000032 :f0a03210: 0=76543210 1=12345678 0000
4180 100: 00000032 :fe811234: 0=76543210 1=12345678 0000
4200 010: 00000032 :fe811234: 0=76543210 1=12345678 0000
4220 001: 00000032 :fe811234: 0=76543210 1=12340000 0000
4240 000: 00000033 :fe811234: 0=76543210 1=12340000 0000
4260 100: 00000033 :f0a15678: 0=76543210 1=12340000 0000
4280 010: 00000033 :f0a15678: 0=76543210 1=12340000 0000
4300 001: 00000033 :f0a15678: 0=76543210 1=12345678 0000
4320 000: 00000034 :f0a15678: 0=76543210 1=12345678 0000
4340 100: 00000034 :f91f0000: 0=76543210 1=12345678 0000
4360 010: 00000034 :f91f0000: 0=76543210 1=12345678 0000
4380 001: 00000018 :f91f0000: 0=76543210 1=12345678 0000
4400 000: 00000018 :f91f0000: 0=76543210 1=12345678 0000
4420 100: 00000018 :f2201234: 0=76543210 1=12345678 0000
4440 010: 00000018 :f2201234: 0=76543210 1=12345678 0000
4460 001: 00000018 :f2201234: 0=76544444 1=12345678 0000
4480 000: 00000019 :f2201234: 0=76544444 1=12345678 0000
4500 100: 00000019 :f8000017: 0=76544444 1=12345678 0000
4520 010: 00000019 :f8000017: 0=76544444 1=12345678 0000
4540 001: 00000030 :f8000017: 0=76544444 1=12345678 0000
4560 000: 00000030 :f8000017: 0=76544444 1=12345678 0000
4580 100: 00000030 :fe807654: 0=76544444 1=12345678 0000
4600 010: 00000030 :fe807654: 0=76544444 1=12345678 0000
4620 001: 00000030 :fe807654: 0=76540000 1=12345678 0000
4640 000: 00000031 :fe807654: 0=76540000 1=12345678 0000
4660 100: 00000031 :f0a03210: 0=76540000 1=12345678 0000
4680 010: 00000031 :f0a03210: 0=76540000 1=12345678 0000
4700 001: 00000031 :f0a03210: 0=76543210 1=12345678 0000
4720 000: 00000032 :f0a03210: 0=76543210 1=12345678 0000
4740 100: 00000032 :fe811234: 0=76543210 1=12345678 0000
4760 010: 00000032 :fe811234: 0=76543210 1=12345678 0000
4780 001: 00000032 :fe811234: 0=76543210 1=12340000 0000
4800 000: 00000033 :fe811234: 0=76543210 1=12340000 0000
4820 100: 00000033 :f0a15678: 0=76543210 1=12340000 0000
4840 010: 00000033 :f0a15678: 0=76543210 1=12340000 0000
4860 001: 00000033 :f0a15678: 0=76543210 1=12345678 0000
4880 000: 00000034 :f0a15678: 0=76543210 1=12345678 0000
4900 100: 00000034 :f91f0000: 0=76543210 1=12345678 0000
4920 010: 00000034 :f91f0000: 0=76543210 1=12345678 0000
4940 001: 0000001a :f91f0000: 0=76543210 1=12345678 0000
4960 000: 0000001a :f91f0000: 0=76543210 1=12345678 0000
4980 100: 0000001a :f2a03f65: 0=76543210 1=12345678 0000
```

```
5000 010: 0000001a :f2a03f65: 0=76543210 1=12345678 0000
5020 001: 0000001a :f2a03f65: 0=7653f2ab 1=12345678 0000
5040 000: 0000001b :f2a03f65: 0=7653f2ab 1=12345678 1000
5060 100: 0000001b :f8000015: 0=7653f2ab 1=12345678 1000
5080 010: 0000001b :f8000015: 0=7653f2ab 1=12345678 1000
5100 001: 00000030 :f8000015: 0=7653f2ab 1=12345678 1000
5120 000: 00000030 :f8000015: 0=7653f2ab 1=12345678 1000
5140 100: 00000030 :fe807654: 0=7653f2ab 1=12345678 1000
5160 010: 00000030 :fe807654: 0=7653f2ab 1=12345678 1000
5180 001: 00000030 :fe807654: 0=76540000 1=12345678 1000
5200 000: 00000031 :fe807654: 0=76540000 1=12345678 1000
5220 100: 00000031 :f0a03210: 0=76540000 1=12345678 1000
5240 010: 00000031 :f0a03210: 0=76540000 1=12345678 1000
5260 001: 00000031 :f0a03210: 0=76543210 1=12345678 1000
5280 000: 00000032 :f0a03210: 0=76543210 1=12345678 0000
5300 100: 00000032 :fe811234: 0=76543210 1=12345678 0000
5320 010: 00000032 :fe811234: 0=76543210 1=12345678 0000
5340 001: 00000032 :fe811234: 0=76543210 1=12340000 0000
5360 000: 00000033 :fe811234: 0=76543210 1=12340000 0000
5380 100: 00000033 :f0a15678: 0=76543210 1=12340000 0000
5400 010: 00000033 :f0a15678: 0=76543210 1=12340000 0000
5420 001: 00000033 :f0a15678: 0=76543210 1=12345678 0000
5440 000: 00000034 :f0a15678: 0=76543210 1=12345678 0000
5460 100: 00000034 :f91f0000: 0=76543210 1=12345678 0000
5480 010: 00000034 :f91f0000: 0=76543210 1=12345678 0000
5500 001: 0000001c :f91f0000: 0=76543210 1=12345678 0000
5520 000: 0000001c :f91f0000: 0=76543210 1=12345678 0000
5540 100: 0000001c :f02003c0: 0=76543210 1=12345678 0000
5560 010: 0000001c :f02003c0: 0=76543210 1=12345678 0000
5580 001: 0000001c :f02003c0: 0=00000200 1=12345678 0000
5600 000: 0000001d :f02003c0: 0=00000200 1=12345678 0000
5620 100: 0000001d :f0a0f5a3: 0=00000200 1=12345678 0000
5640 010: 0000001d :f0a0f5a3: 0=00000200 1=12345678 0000
5660 001: 0000001d :f0a0f5a3: 0=0000f7a3 1=12345678 0000
5680 000: 0000001e :f0a0f5a3: 0=0000f7a3 1=12345678 0000
5700 100: 0000001e :f1203f06: 0=0000f7a3 1=12345678 0000
5720 010: 0000001e :f1203f06: 0=0000f7a3 1=12345678 0000
5740 001: 0000001e :f1203f06: 0=0000c8a5 1=12345678 0000
5760 000: 0000001f :f1203f06: 0=0000c8a5 1=12345678 0000
5780 100: 0000001f :b8800000: 0=0000c8a5 1=12345678 0000
5800 010: 0000001f :b8800000: 0=0000c8a5 1=12345678 0000
5820 001: 0000001f :b8800000: 0=0000c8a5 1=12345678 0000
5840 000: 00000020 :b8800000: 0=0000c8a5 1=12345678 0000
5860 100: 00000020 :f0400800: 0=0000c8a5 1=12345678 0000
5880 010: 00000020 :f0400800: 0=0000c8a5 1=12345678 0000
5900 001: 00000020 :f0400800: 0=ffffbfdf 1=12345678 0000
5920 000: 00000021 :f0400800: 0=ffffbfdf 1=12345678 0100
5940 100: 00000021 :f800000f: 0=ffffbfdf 1=12345678 0100
5960 010: 00000021 :f800000f: 0=ffffbfdf 1=12345678 0100
5980 001: 00000030 :f800000f: 0=ffffbfdf 1=12345678 0100
6000 000: 00000030 :f800000f: 0=ffffbfdf 1=12345678 0100
6020 100: 00000030 :fe807654: 0=ffffbfdf 1=12345678 0100
6040 010: 00000030 :fe807654: 0=ffffbfdf 1=12345678 0100
6060 001: 00000030 :fe807654: 0=76540000 1=12345678 0100
```

```
6080 000: 00000031 :fe807654: 0=76540000 1=12345678 0100
6100 100: 00000031 :f0a03210: 0=76540000 1=12345678 0100
6120 010: 00000031 :f0a03210: 0=76540000 1=12345678 0100
6140 001: 00000031 :f0a03210: 0=76543210 1=12345678 0100
6160 000: 00000032 :f0a03210: 0=76543210 1=12345678 0000
6180 100: 00000032 :fe811234: 0=76543210 1=12345678 0000
6200 010: 00000032 :fe811234: 0=76543210 1=12345678 0000
6220 001: 00000032 :fe811234: 0=76543210 1=12340000 0000
6240 000: 00000033 :fe811234: 0=76543210 1=12340000 0000
6260 100: 00000033 :f0a15678: 0=76543210 1=12340000 0000
6280 010: 00000033 :f0a15678: 0=76543210 1=12340000 0000
6300 001: 00000033 :f0a15678: 0=76543210 1=12345678 0000
6320 000: 00000034 :f0a15678: 0=76543210 1=12345678 0000
6340 100: 00000034 :f91f0000: 0=76543210 1=12345678 0000
6360 010: 00000034 :f91f0000: 0=76543210 1=12345678 0000
6380 001: 00000022 :f91f0000: 0=76543210 1=12345678 0000
6400 000: 00000022 :f91f0000: 0=76543210 1=12345678 0000
6420 100: 00000022 :f0c00800: 0=76543210 1=12345678 0000
6440 010: 00000022 :f0c00800: 0=76543210 1=12345678 0000
6460 001: 00000022 :f0c00800: 0=898b8987 1=12345678 0000
6480 000: 00000023 :f0c00800: 0=898b8987 1=12345678 0100
6500 100: 00000023 :f800000d: 0=898b8987 1=12345678 0100
6520 010: 00000023 :f800000d: 0=898b8987 1=12345678 0100
6540 001: 00000030 :f800000d: 0=898b8987 1=12345678 0100
6560 000: 00000030 :f800000d: 0=898b8987 1=12345678 0100
6580 100: 00000030 :fe807654: 0=898b8987 1=12345678 0100
6600 010: 00000030 :fe807654: 0=898b8987 1=12345678 0100
6620 001: 00000030 :fe807654: 0=76540000 1=12345678 0100
6640 000: 00000031 :fe807654: 0=76540000 1=12345678 0100
6660 100: 00000031 :f0a03210: 0=76540000 1=12345678 0100
6680 010: 00000031 :f0a03210: 0=76540000 1=12345678 0100
6700 001: 00000031 :f0a03210: 0=76543210 1=12345678 0100
6720 000: 00000032 :f0a03210: 0=76543210 1=12345678 0000
6740 100: 00000032 :fe811234: 0=76543210 1=12345678 0000
6760 010: 00000032 :fe811234: 0=76543210 1=12345678 0000
6780 001: 00000032 :fe811234: 0=76543210 1=12340000 0000
6800 000: 00000033 :fe811234: 0=76543210 1=12340000 0000
6820 100: 00000033 :f0a15678: 0=76543210 1=12340000 0000
6840 010: 00000033 :f0a15678: 0=76543210 1=12340000 0000
6860 001: 00000033 :f0a15678: 0=76543210 1=12345678 0000
6880 000: 00000034 :f0a15678: 0=76543210 1=12345678 0000
6900 100: 00000034 :f91f0000: 0=76543210 1=12345678 0000
6920 010: 00000034 :f91f0000: 0=76543210 1=12345678 0000
6940 001: 00000024 :f91f0000: 0=76543210 1=12345678 0000
6960 000: 00000024 :f91f0000: 0=76543210 1=12345678 0000
6980 100: 00000024 :f1400800: 0=76543210 1=12345678 0000
7000 010: 00000024 :f1400800: 0=76543210 1=12345678 0000
7020 001: 00000024 :f1400800: 0=9b9f9b97 1=12345678 0000
7040 000: 00000025 :f1400800: 0=9b9f9b97 1=12345678 0100
7060 100: 00000025 :fe400000: 0=9b9f9b97 1=12345678 0100
7080 010: 00000025 :fe400000: 0=00000000 1=12345678 0100
7100 001: 00000025 :fe400000: 0=00000000 1=12345678 0100
7120 000: 00000026 :fe400000: 0=00000000 1=12345678 0100
7140 100: 00000026 :fe000000: 0=00000000 1=12345678 0100
```

```
7160 010: 00000026 :fe000000: 0=00000000 1=12345678 0100
7180 001: 00000026 :fe000000: 0=00000001 1=12345678 0100
7200 000: 00000027 :fe000000: 0=00000001 1=12345678 0000
7220 100: 00000027 :fe000000: 0=00000001 1=12345678 0000
7240 010: 00000027 :fe000000: 0=00000001 1=12345678 0000
7260 001: 00000027 :fe000000: 0=00000002 1=12345678 0000
7280 000: 00000028 :fe000000: 0=00000002 1=12345678 0000
7300 100: 00000028 :f2a00002: 0=00000002 1=12345678 0000
7320 010: 00000028 :f2a00002: 0=00000002 1=12345678 0000
7340 001: 00000028 :f2a00002: 0=00000000 1=12345678 0000
7360 000: 00000029 :f2a00002: 0=00000000 1=12345678 1001
7380 100: 00000029 :a8800003: 0=00000000 1=12345678 1001
7400 010: 00000029 :a8800003: 0=00000000 1=12345678 1001
7420 001: 00000029 :a8800003: 0=00000000 1=12345678 1001
7440 000: 0000002a :a8800003: 0=00000000 1=12345678 1001
7460 100: 0000002a :b8800002: 0=00000000 1=12345678 1001
7480 010: 0000002a :b8800002: 0=00000000 1=12345678 1001
7500 001: 0000002c :b8800002: 0=00000000 1=12345678 1001
7520 000: 0000002c :b8800002: 0=00000000 1=12345678 1001
7540 100: 0000002c :b8000004: 0=00000000 1=12345678 1001
7560 010: 0000002c :b8000004: 0=00000000 1=12345678 1001
7580 001: 00000030 :b8000004: 0=00000000 1=12345678 1001
7600 000: 00000030 :b8000004: 0=00000000 1=12345678 1001
7620 100: 00000030 :fe807654: 0=00000000 1=12345678 1001
7640 010: 00000030 :fe807654: 0=00000000 1=12345678 1001
7660 001: 00000030 :fe807654: 0=76540000 1=12345678 1001
7680 000: 00000031 :fe807654: 0=76540000 1=12345678 1001
7700 100: 00000031 :f0a03210: 0=76540000 1=12345678 1001
7720 010: 00000031 :f0a03210: 0=76540000 1=12345678 1001
7740 001: 00000031 :f0a03210: 0=76543210 1=12345678 1001
7760 000: 00000032 :f0a03210: 0=76543210 1=12345678 0000
7780 100: 00000032 :fe811234: 0=76543210 1=12345678 0000
7800 010: 00000032 :fe811234: 0=76543210 1=12345678 0000
7820 001: 00000032 :fe811234: 0=76543210 1=12340000 0000
7840 000: 00000033 :fe811234: 0=76543210 1=12340000 0000
7860 100: 00000033 :f0a15678: 0=76543210 1=12340000 0000
7880 010: 00000033 :f0a15678: 0=76543210 1=12340000 0000
7900 001: 00000033 :f0a15678: 0=76543210 1=12345678 0000
7920 000: 00000034 :f0a15678: 0=76543210 1=12345678 0000
7940 100: 00000034 :f91f0000: 0=76543210 1=12345678 0000
7960 010: 00000034 :f91f0000: 0=76543210 1=12345678 0000
7980 001: 0000002d :f91f0000: 0=76543210 1=12345678 0000
8000 000: 0000002d :f91f0000: 0=76543210 1=12345678 0000
8020 100: 0000002d :00000000: 0=76543210 1=12345678 0000
8040 010: 0000002d :00000000: 0=76543210 1=12345678 0000
8060 001: 0000002d :00000000: 0=76543210 1=12345678 0000
8080 000: 0000002e :00000000: 0=76543210 1=12345678 0000
8100 100: 0000002e :00000000: 0=76543210 1=12345678 0000
8120 010: 0000002e :00000000: 0=76543210 1=12345678 0000
8140 001: 0000002e :00000000: 0=76543210 1=12345678 0000
8160 000: 0000002f :00000000: 0=76543210 1=12345678 0000
8180 100: 0000002f :f8800007: 0=76543210 1=12345678 0000
8200 010: 0000002f :f8800007: 0=76543210 1=12345678 0000
8220 001: 00000036 :f8800007: 0=76543210 1=12345678 0000
```

```
8240 000: 00000036 :f8800007: 0=76543210 1=12345678 0000
8260 100: 00000036 :f87ffffa: 0=76543210 1=12345678 0000
8280 010: 00000036 :f87ffffa: 0=76543210 1=12345678 0000
8300 001: 00000030 :f87ffffa: 0=76543210 1=12345678 0000
8320 000: 00000030 :f87ffffa: 0=76543210 1=12345678 0000
8340 100: 00000030 :fe807654: 0=76543210 1=12345678 0000
8360 010: 00000030 :fe807654: 0=76543210 1=12345678 0000
8380 001: 00000030 :fe807654: 0=76540000 1=12345678 0000
8400 000: 00000031 :fe807654: 0=76540000 1=12345678 0000
8420 100: 00000031 :f0a03210: 0=76540000 1=12345678 0000
8440 010: 00000031 :f0a03210: 0=76540000 1=12345678 0000
8460 001: 00000031 :f0a03210: 0=76543210 1=12345678 0000
8480 000: 00000032 :f0a03210: 0=76543210 1=12345678 0000
8500 100: 00000032 :fe811234: 0=76543210 1=12345678 0000
8520 010: 00000032 :fe811234: 0=76543210 1=12345678 0000
8540 001: 00000032 :fe811234: 0=76543210 1=12340000 0000
8560 000: 00000033 :fe811234: 0=76543210 1=12340000 0000
8580 100: 00000033 :f0a15678: 0=76543210 1=12340000 0000
8600 010: 00000033 :f0a15678: 0=76543210 1=12340000 0000
8620 001: 00000033 :f0a15678: 0=76543210 1=12345678 0000
8640 000: 00000034 :f0a15678: 0=76543210 1=12345678 0000
8660 100: 00000034 :f91f0000: 0=76543210 1=12345678 0000
8680 010: 00000034 :f91f0000: 0=76543210 1=12345678 0000
8700 001: 00000037 :f91f0000: 0=76543210 1=12345678 0000
8720 000: 00000037 :f91f0000: 0=76543210 1=12345678 0000
8740 100: 00000037 :fe400000: 0=76543210 1=12345678 0000
8760 010: 00000037 :fe400000: 0=00000000 1=12345678 0000
8780 001: 00000037 :fe400000: 0=00000000 1=12345678 0000
8800 000: 00000038 :fe400000: 0=00000000 1=12345678 0000
8820 100: 00000038 :fe000000: 0=00000000 1=12345678 0000
8840 010: 00000038 :fe000000: 0=00000000 1=12345678 0000
8860 001: 00000038 :fe000000: 0=00000001 1=12345678 0000
8880 000: 00000039 :fe000000: 0=00000001 1=12345678 0000
8900 100: 00000039 :f2a00002: 0=00000001 1=12345678 0000
8920 010: 00000039 :f2a00002: 0=00000001 1=12345678 0000
8940 001: 00000039 :f2a00002: 0=ffffffff 1=12345678 0000
8960 000: 0000003a :f2a00002: 0=ffffffff 1=12345678 0100
8980 100: 0000003a :b8800003: 0=ffffffff 1=12345678 0100
9000 010: 0000003a :b8800003: 0=ffffffff 1=12345678 0100
9020 001: 0000003a :b8800003: 0=ffffffff 1=12345678 0100
9040 000: 0000003b :b8800003: 0=ffffffff 1=12345678 0100
9060 100: 0000003b :a8800002: 0=ffffffff 1=12345678 0100
9080 010: 0000003b :a8800002: 0=ffffffff 1=12345678 0100
9100 001: 0000003d :a8800002: 0=ffffffff 1=12345678 0100
9120 000: 0000003d :a8800002: 0=ffffffff 1=12345678 0100
9140 100: 0000003d :fee00013: 0=ffffffff 1=12345678 0100
9160 010: 0000003d :fee00013: 0=ffffffff 1=12345678 0100
9180 001: 0000003d :fee00013: 0=00000013 1=12345678 0100
9200 000: 0000003e :fee00013: 0=00000013 1=12345678 0100
9220 100: 0000003e :fee10003: 0=00000013 1=12345678 0100
9240 010: 0000003e :fee10003: 0=00000013 1=12345678 0100
9260 001: 0000003e :fee10003: 0=00000013 1=00000003 0100
9280 000: 0000003f :fee10003: 0=00000013 1=00000003 0100
9300 100: 0000003f :f4000800: 0=00000013 1=00000003 0100
```

```
 9320 010: 0000003f :f4000800: 0=00000013 1=00000003 0100
 9340 001: 0000003f :f4000800: 0=00000098 1=00000003 0100
 9360 000: 00000040 :f4000800: 0=00000098 1=00000003 0000
 9380 100: 00000040 :fe210000: 0=00000098 1=00000003 0000
 9400 010: 00000040 :fe210000: 0=00000098 1=00000003 0000
 9420 001: 00000040 :fe210000: 0=00000098 1=00000002 0000
 9440 000: 00000041 :fe210000: 0=00000098 1=00000002 0000
 9460 100: 00000041 :f4800800: 0=00000098 1=00000002 0000
 9480 010: 00000041 :f4800800: 0=00000098 1=00000002 0000
 9500 001: 00000041 :f4800800: 0=00000026 1=00000002 0000
 9520 000: 00000042 :f4800800: 0=00000026 1=00000002 0000
 9540 100: 00000042 :f4a00001: 0=00000026 1=00000002 0000
 9560 010: 00000042 :f4a00001: 0=00000026 1=00000002 0000
 9580 001: 00000042 :f4a00001: 0=00000004 1=00000002 0000
 9600 000: 00000043 :f4a00001: 0=00000004 1=00000002 0000
 9620 100: 00000043 :fe80f234: 0=00000004 1=00000002 0000
 9640 010: 00000043 :fe80f234: 0=00000004 1=00000002 0000
 9660 001: 00000043 :fe80f234: 0=f2340000 1=00000002 0000
 9680 000: 00000044 :fe80f234: 0=f2340000 1=00000002 0000
 9700 100: 00000044 :f4c00800: 0=f2340000 1=00000002 0000
 9720 010: 00000044 :f4c00800: 0=f2340000 1=00000002 0000
 9740 001: 00000044 :f4c00800: 0=f2340000 1=00000002 0000
 9760 000: 00000045 :f4c00800: 0=f2340000 1=00000002 0100
 9780 100: 00000045 :f5200003: 0=f2340000 1=00000002 0100
 9800 010: 00000045 :f5200003: 0=f2340000 1=00000002 0100
 9820 001: 00000045 :f5200003: 0=91a00007 1=00000002 0100
 9840 000: 00000046 :f5200003: 0=91a00007 1=00000002 0100
 9860 100: 00000046 :f5a00002: 0=91a00007 1=00000002 0100
 9880 010: 00000046 :f5a00002: 0=91a00007 1=00000002 0100
 9900 001: 00000046 :f5a00002: 0=e4680001 1=00000002 0100
 9920 000: 00000047 :f5a00002: 0=e4680001 1=00000002 0100
 9940 100: 00000047 :f5000800: 0=e4680001 1=00000002 0100
 9960 010: 00000047 :f5000800: 0=e4680001 1=00000002 0100
 9980 001: 00000047 :f5000800: 0=e4680001 1=00000002 0100
10000 000: 00000048 :f5000800: 0=e4680001 1=00000002 0100
10020 100: 00000048 :fe010000: 0=e4680001 1=00000002 0100
10040 010: 00000048 :fe010000: 0=e4680001 1=00000002 0100
10060 001: 00000048 :fe010000: 0=e4680001 1=00000003 0100
10080 000: 00000049 :fe010000: 0=e4680001 1=00000003 0000
10100 100: 00000049 :f5800800: 0=e4680001 1=00000003 0000
10120 010: 00000049 :f5800800: 0=e4680001 1=00000003 0000
10140 001: 00000049 :f5800800: 0=3c8d0000 1=00000003 0000
10160 000: 0000004a :f5800800: 0=3c8d0000 1=00000003 0000
10180 100: 0000004a :fee00013: 0=3c8d0000 1=00000003 0000
10200 010: 0000004a :fee00013: 0=3c8d0000 1=00000003 0000
10220 001: 0000004a :fee00013: 0=00000013 1=00000003 0000
10240 000: 0000004b :fee00013: 0=00000013 1=00000003 0000
10260 100: 0000004b :fee1000b: 0=00000013 1=00000003 0000
10280 010: 0000004b :fee1000b: 0=00000013 1=00000003 0000
10300 001: 0000004b :fee1000b: 0=00000013 1=0000000b 0000
10320 000: 0000004c :fee1000b: 0=00000013 1=0000000b 0000
10340 100: 0000004c :f6000800: 0=00000013 1=0000000b 0000
10360 010: 0000004c :f6000800: 0=00000013 1=0000000b 0000
10380 001: 0000004c :f6000800: 0=000000d1 1=0000000b 0000
```

```
10400 000: 0000004d :f6000800: 0=000000d1 1=0000000b x000
10420 100: 0000004d :fee00013: 0=000000d1 1=0000000b x000
10440 010: 0000004d :fee00013: 0=000000d1 1=0000000b x000
10460 001: 0000004d :fee00013: 0=00000013 1=0000000b x000
10480 000: 0000004e :fee00013: 0=00000013 1=0000000b x000
10500 100: 0000004e :fee10003: 0=00000013 1=0000000b x000
10520 010: 0000004e :fee10003: 0=00000013 1=0000000b x000
10540 001: 0000004e :fee10003: 0=00000013 1=00000003 x000
10560 000: 0000004f :fee10003: 0=00000013 1=00000003 x000
10580 100: 0000004f :f7000800: 0=00000013 1=00000003 x000
10600 010: 0000004f :f7000800: 0=00000013 1=00000003 x000
10620 001: 0000004f :f7000800: 0=00000006 1=00000003 x000
10640 000: 00000050 :f7000800: 0=00000006 1=00000003 x000
10660 100: 00000050 :fe80f324: 0=00000006 1=00000003 x000
10680 010: 00000050 :fe80f324: 0=00000006 1=00000003 x000
10700 001: 00000050 :fe80f324: 0=f3240000 1=00000003 x000
10720 000: 00000051 :fe80f324: 0=f3240000 1=00000003 x000
10740 100: 00000051 :f0a00abc: 0=f3240000 1=00000003 x000
10760 010: 00000051 :f0a00abc: 0=f3240000 1=00000003 x000
10780 001: 00000051 :f0a00abc: 0=f3240abc 1=00000003 x000
10800 000: 00000052 :f0a00abc: 0=f3240abc 1=00000003 x100
10820 100: 00000052 :fe410000: 0=f3240abc 1=00000003 x100
10840 010: 00000052 :fe410000: 0=f3240abc 1=00000000 x100
10860 001: 00000052 :fe410000: 0=f3240abc 1=00000000 x100
10880 000: 00000053 :fe410000: 0=f3240abc 1=00000000 x100
10900 100: 00000053 :ff800800: 0=f3240abc 1=00000000 x100
10920 010: 00000053 :ff800800: 0=f3240abc 1=00000000 x100
10940 001: 00000053 :ff800800: 0=f3240abc 1=00000000 x100
10960 000: 00000054 :ff800800: 0=f3240abc 1=00000000 x100
10980 100: 00000054 :fe400000: 0=f3240abc 1=00000000 x100
11000 010: 00000054 :fe400000: 0=00000000 1=00000000 x100
11020 001: 00000054 :fe400000: 0=00000000 1=00000000 x100
11040 000: 00000055 :fe400000: 0=00000000 1=00000000 x100
11060 100: 00000055 :ff000800: 0=00000000 1=00000000 x100
11080 010: 00000055 :ff000800: 0=00000000 1=00000000 x100
11100 001: 00000055 :ff000800: 0=f3240abc 1=00000000 x100
11120 000: 00000056 :ff000800: 0=f3240abc 1=00000000 x100
11140 100: 00000056 :f8800000: 0=f3240abc 1=00000000 x100
11160 010: 00000056 :f8800000: 0=f3240abc 1=00000000 x100
11180 001: 00000056 :f8800000: 0=f3240abc 1=00000000 x100
11200 000: 00000056 :f8800000: 0=f3240abc 1=00000000 x100
11220 100: 00000056 :f8800000: 0=f3240abc 1=00000000 x100
11240 010: 00000056 :f8800000: 0=f3240abc 1=00000000 x100
11260 001: 00000056 :f8800000: 0=f3240abc 1=00000000 x100
11280 000: 00000056 :f8800000: 0=f3240abc 1=00000000 x100
11300 100: 00000056 :f8800000: 0=f3240abc 1=00000000 x100
11320 010: 00000056 :f8800000: 0=f3240abc 1=00000000 x100
11340 001: 00000056 :f8800000: 0=f3240abc 1=00000000 x100
11360 000: 00000056 :f8800000: 0=f3240abc 1=00000000 x100
11380 100: 00000056 :f8800000: 0=f3240abc 1=00000000 x100
```

11.1 ECE5480 Class Instruction Matrix

For the ECE5480 class, the required instructions to execute were a subset if instructions from the MIPS processor:

1) **arithmetic:** addu, subu, addiu, mult, div
2) **logical:** and, or, xor, andi, ori, xori, lui
3) **shift:** sll, sra, srl
4) **compare:** slt, slti, sltu, sltiu
5) **control:** beq, bne, j, jr, jal
6) **data transfer:** lw, sw

The Pioneer-2 is a very different design than the MIPS is, but most of the instructions have a direct correlation. For example, the addui instruction from the MIPS is the same instruction as the Pioneer-2 Rx=Rx+Ry instruction. Table 11.1 shows the translation between the two instruction sets, and the address and time reference to find the instruction being executed in the test program.

MIPS	Pioneer-2	location	$time
ADDU	Rx = Rx + Ry	00000003	340
SUBU	Rx = Rx - Ry	00000008	740
ADDIU	Rx = Rx + ####	00000018	
MULT	Rx = Rx * Ry	0000004C	
DIV	Rx = Rx / Ry	0000004F	
AND	Rx = Rx AND Ry	00000010	2180
OR	Rx = Rx OR Ry	00000012	2720
XOR	Rx = Rx XOR Ry	00000014	3300
ANDI	Rx = Rx AND ####	0000001C	5540
ORI	Rx = Rx OR ####	0000001D	5620
XORI	Rx = Rx XOR ####	0000001E	5700
LUI	RxH = ####	00000001	180
SLL	LSHIFT Rx BY Ry	0000003F	9300
SRA	ASHIFT Rx BY Ry	00000044	9700
SRL	RSHIFT Rx BY Ry	00000041	9460
SLT SLTI SLTU SLTIU	See documentation		
BEQ	IF EQ JUMP	001F, 002A	5780, 7460
BNE	IF NE JUMP	0029, 003B	7380, 9060
J	JUMP	002F, 0056	8180, 11140
JR	JR (or RTS)	00000034	1380
JAL	CALL	0000000B	980
LW	Rx=DM[Ry],offset	00000053	10900
SW	DM[Rx],offset=Ry	00000055	11060

Table 11.1: MIPS/Pioneer-2 Translation

Because the structure of the Pioneer-2 instruction set uses a conditional predicate as part of each instruction, the SLT instruction (including SLTI, SLTU, and SLTUI) from the MIPS processor have been dropped in favor of other instructions. However, there are some other instructions that mimic the SLT instructions. The SLT collections of instructions are a combination of subtraction without writeback and set/unset words. The Pioneer-2 processor would use a CMP instruction (which also has an immediate mode option) could be followed by a conditional LLI[2]:

[2]The LLI is an additional instruction added to the Pioneer-2, acting similar to the LUI, but effecting the low halfword of the register.

```
        CHECK R0 - R1           ;If R0 < R1, N flag is set, but R0 is not changed
IF LT   R2L = 1                 ;If N flag is set, R2=0x00000001
```

Because the CMP (or “CHECK”) instruction can have either a register or immediate value, these should be used to emulate the SLT and SLTI. Likewise, because the carry flag is used for unsigned numbers and the overflow flag is used for signed numbers, these two flags could be used to discriminate bwteen the SLT(I) and the SLTU(I) instructions.

11.2 Additional Instructions

Several instructions were added to the Pioneer-2processor, beyond the required instruction set. Table 11.2 highlights what these instructions are, where they were used in the test program, and what simulated time these instructions were demonstrated.

Instruction	location	time
CLC (clear carry)	00000000	100
PASS (set N, Z flags)	00000002	260
IF C predicate for ADD	00000005	500
IF NC predicate for PASS	00000006	580
NEG (2’s complement)	0000000C	1460
NOP (“IF NEVER”)	0000000F	2100
NOT (1’s compliment in ALU)	00000016	3860
SUBI	0000001A	4980
NAND[3]	00000020	5860
NOR	00000022	6420
XNOR	00000024	6980
CLEAR	00000025	7060
INC	00000026	7140
IF Z predicate for CALL	0000002C	7540
LLI (lower halfword load)	0000003D	9140
DEC	00000040	9380
RSHIFT immediate	00000042	9540
LROTATE immediate	00000045	9780
RROTATE immediate	00000046	9860
LROTATE BY register	00000047	9940
RROTATE BY register	00000049	10100

Table 11.2: Additional Instructions

11.3 Outstanding Issues

There are a number of outstanding issues to be addressed in future versions of the Pioneer-2:

1) STC (set carry) flag instruction is broken
2) Many modules are still modeled in behavioral mode instead of structural
3) Special Purpose registers are not yet accessible

4) Carry flag does not work as desired in shift and rotate instructions
5) Address Generator is not available
6) Stack frome instructions not available (ENTER, LEAVE, PUSH, POP)
7) MOVE instruction (general and special purpose registers) not available
8) Priviledged Mode not available
9) Interrupt and Exception handling not available
10) Memory Management not available
11) Hardware interfaces (DMA, HOLD, Wait state) not available
12) Floating Point Unit not available

However, none of the missing components were required for this version of the Pioneer-2.

REFERENCES

"The TTL Data Book, Volume 2," Texas Instruments Incorporated, 1985

"ADSP-2101 User's Manual, Architecture," Analog Devices Incorporated, 1990

"Intel 386SX Microprocessor, Programmer's Reference Manual," Intel Literature Sales, 1989