ECPE 173 – Spring 2013
Lab Project 4

In this lab project, we will extend our ALU to a basic processor supporting most instructions except branches and jumps. We will call this processor Beta. In addition to existing documentation (including the Lab Project 3 handout), you may want to look at the *Summary of Beta Instruction Formats* and *Beta Documentation*.

Our basic design will look like Figure 1.

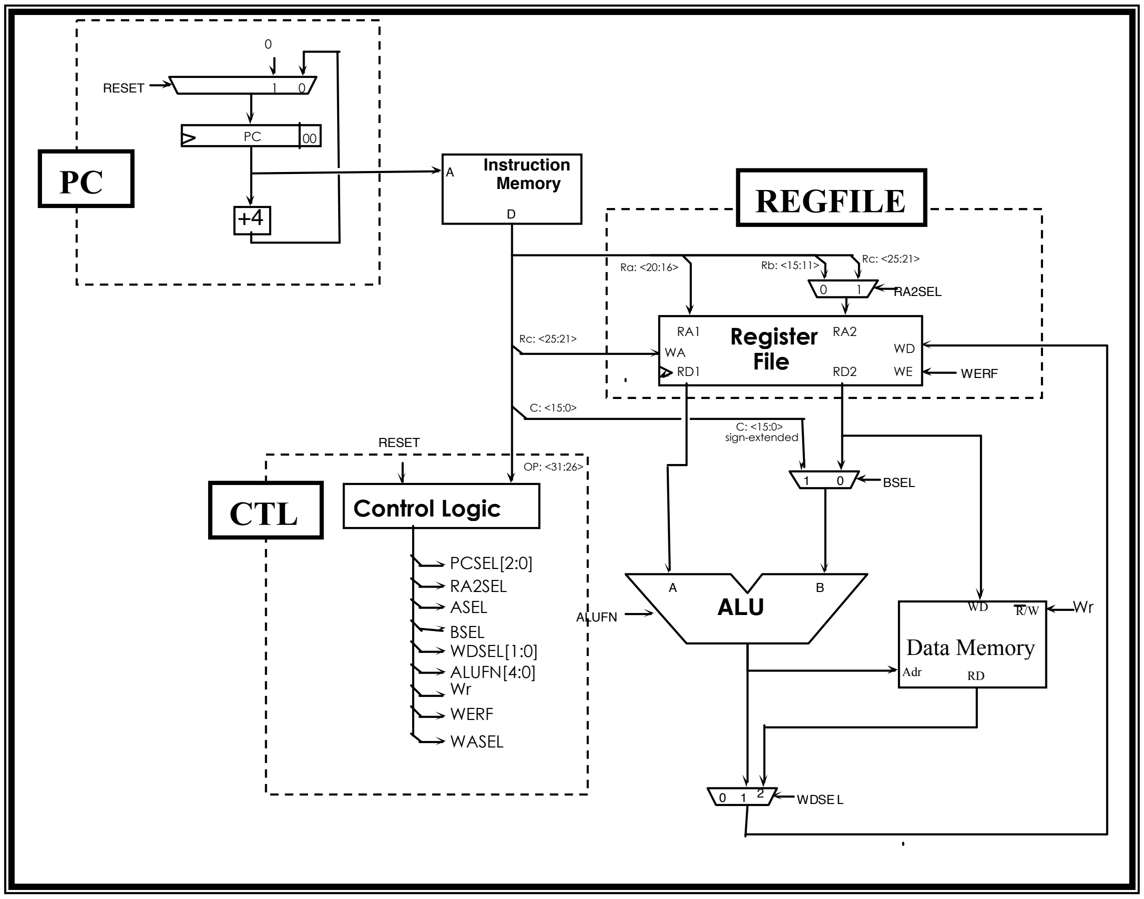

Figure 1: Basic Beta Processor

1. Everyone needs to implement his or her own Lab Project 4. If you collaborate with anyone, you need to follow the collaboration policy. No collaboration on the writing is allowed – any duplicate sentences/paragraphs will result in a zero.

Start a Word document report for this project. Create appropriate document headings and write a short Problem section describing the goals of this lab (in your own words – not mine!).

2. Review the instruction formats for the Beta and Figure 1. Determine the differences between the Beta and our standard MIPS processor. In a new section in your Word document, describe them in a paragraph.

3. Create a new folder for this lab and copy *lab3.jsim* there. Rename it to *lab4.jsim*. Download checkoff files from Sakai->Resources->Lab Projects->Project 4.

4. Design the PC first. We will include a 32-bit multiplexer to select based on an active-high RESET whether to reset the system and set PC=0x0 or to set PC=PC+4. PC itself is a rising-edge triggered register. To implement the register for the PC, we will use a dreg component. The primary subcircuit needs to be defined as:

```
.subckt pc clk reset ia[31:0]
        … implementation here …
.ends
```

To verify your implementation, use:

```
.include “lab4pc.jsim"
```

In your Word document, create a PC section. Describe your design in your own words, include the relevant code, and a screenshot of the functional design.

4. Design the register. The register is a 3-port memory. To instantiate a memory, you use a netlist similar to:

```
Xregfile
+ vdd 0 0 ra[4:0] adata[31:0]          // A read port
+ vdd 0 0 ra2mux[4:0] bdata[31:0]      // B read port
+ 0 clk werf rc[4:0] wdata[31:0]       // write port
+ $memory width=32 nlocations=32
```

Further information on memory components is located in Appendix A of this handout.

For our Beta, $zero is located at register 31. The memory does not know this implicitly; you need to explicitly design the register logic to deal with this. You will also have to design logic to choose the correct source address for the B read port.

To implement the register subcircuit, use the following definition model:

```
.subckt regfile clk werf ra2sel ra[4:0] rb[4:0] rc[4:0]
+ wdata[31:0] radata[31:0] rbdata[31:0]
        … implementation here …
.ends
```

To verify your implementation, use:

```
.include “lab4reg.jsim"
```

In your Word document, create a Register File section. Describe your design in your own words, include the relevant code, and a screenshot of the functional design.

4. Design the control logic. You can either implement it using gate-level design or a ROM. ROM design is described in Appendix B.

As of now, your design should support:

    LD, ST, ADD, SUB, CMPEQ, CMPLT, CMPLE, AND, OR, XOR, XNOR,
    SHL, SHR, SRA, ADDC, SUBC, CMPEQC, CMPLTC, CMPLEC, ANDC,
    ORC, XORC, XNORC, SHLC, SHRC, SRAC

Any instructions not in this list should be considered NOPs (not operations), which you can achieve by setting WERF to 0 (WERF=write enable register file).

You need to be especially careful in designing your write enable for main memory. When the system is reset, WR should be 0 (memory writes should not be enabled). In addition to this, you need to define MOE (memory output enable), which will go high on memory reads. Figure 1 outlines all other control signals needed.

To implement the control subcircuit, use the following definition model:

```
.subckt ctl reset id[31:26] ra2sel bsel alufn[4:0] wdsel[1:0] werf
+ moe wr
        … implementation here …
.ends
```

To verify your implementation, use:

```
.include “lab4ctl.jsim"
```

In your Word document, create a Control section. Describe your design in your own words, include the relevant code, and a screenshot of the functional design.

5. Now, you will pull all of the pieces together. The checkoff file will instantiate the instruction and data memory units. You need to supply the necessary address, data, and control signals:

    ia[31:0] – address of next instruction to execute (OUTPUT)
    id[31:0] – data from instruction memory, instruction to execute (INPUT)
    ma[31:0] – address of data memory to read or write (OUTPUT)
    moe – set to 1 to enable read (OUTPUT)
    mrd[31:0] – memory read data (INPUT)
    wr – set to 1 to write data to memory (OUTPUT)
    mwd[31:0] – data to write into memory (OUTPUT)

Your Beta must generate everything that is labeled OUTPUT.

In addition to memory, you need to add logic to support sign-extension of I-type instructions and the mux to select write destination for the register.

To implement the Beta subcircuit, use the following definition model:

```
.subckt beta clk reset ia[31:0] id[31:0] ma[31:0] moe mrd[31:0] wr
+ mwd[31:0]
        … implementation here …
.ends
```

To verify your implementation, use:

```
.include "lab4beta.jsim"
```

In your Word document, create a Beta section. Describe your design in your own words, include the relevant code, and a screenshot of the functional design.

6. Turn in your Word file and complete JSIM code via Sakai.

Appendix A – Memory Components

We'll be using a new component in this lab: a multi-port memory. JSim has a built-in memory device that can be used to model memories with a specified width and number of locations, and with one or more ports. Each port has 3 control signals and the specified number of address and data wires. You can instantiate a memory device in your circuit with a statement of the form

X*id ports…* $memory width=*w* nlocations=*nloc options…*

The width and nlocations properties must be supplied: *w* specifies the width of each memory location in bits and must be between 1 and 32. `nloc` specifies the number of memory locations and must be between 1 and 2^{20}. All the ports of a memory access the same internal storage, but each port operates independently. Each `port` specification is a list of nodes:

oe clk wen $a_{naddr-1}$ … a_0 d_{w-1} … d_0

where

oe is the output enable input for a read port. When 1, data is driven onto the data pins; when 0, the output pins are not driven by this memory port. If this port is only a write port, connect this terminal to the ground node "0". If the port is only a read port and should always be enabled, connect this terminal to the power supply node "vdd".

clk is the clock input for write ports. When wen=1, data from the data terminals is written into the memory on the rising edge of clk. If this port is only a read port, connect this terminal to the ground node "0".

wen is the write enable input for write ports. See the description of "clk" for details about the write operation. If this port is only a read port, connect this terminal to the ground node "0".

$a_{naddr-1}$ … a_0 are the address inputs, listed most significant bit first. The values of these terminals are used to compute the address of the memory location to be read or written. The number of address terminals is determined from the number of locations in the memory: naddr = ceiling($\log_2$(*nloc*)). When the number of locations in a memory isn't exactly a power of 2, reads that refer to non-existent locations return "X" and writes to non-existent locations have no effect.

d_{w-1} … d_0 are the data inputs/tristate outputs, listed most significant bit first.

By specifying one of the following options it is possible to specify the initial contents of a memory (if not specified, the memory is initialized to all X's):

`file="`*`filename`*`"`

The memory is initialized, location-by-location, from bytes in the file. Data is assumed to be in a binary little-endian format, using ceiling(w/8) bytes of file data per memory location. Bits 0 through 7 of the first file byte are used to initialize bits 0 through 7 of memory location 0, bits 0 through 7 of the second file byte are used to initialize bits 8 through 15 of memory location 0, and so on. When all the bits in a memory location have been filled, any bits remaining in the current file byte are discarded and then the process continues with the next memory location. In particular, the ".bin" files produced by BSim can be used to initialize JSim memories. For example, the following statement would create a 1024-location 32-bit memory with three ports: 2 read ports and 1 one write port. The memory is initialized from the BSim output file "foo.bin".

```
Xmem
+ vdd 0 0 ia[11:2] id[31:0]    // (read) instruction data
+ vdd 0 0 ma[11:2] mrd[31:0]   // (read) program data (LDs)
+ 0 clk wr ma[11:2] mwd[31:0]  // (write) program data (STs)
+ $memory width=32 nlocations=1024
+ file="foo.bin"
```

`contents=(` *`data…`* `)`

The memory is initialized, location-by-location, from the data values given in the list. The least significant bit (bit 0) of a value is used to initialize bit 0 of a memory location, bit 1 of a value is used to initialize bit 1 of a memory location, etc. For example, to enter the short test program ADDC(R31,1,R0); ADDC(R31,2,R1); ADD(R0,R1,R2) one might specify:

```
Xmem
+ vdd 0 0 ia[11:2] id[31:0]    // (read) instruction data
+ vdd 0 0 ma[11:2] mrd[31:0]   // (read) program data (LDs)
+ 0 clk wr ma[11:2] mwd[31:0]  // (write) program data (STs)
+ $memory width=32 nlocations=1024
+ contents=(0xC01F0001 0xC03F0002 0x80400800)
```

Initialized memories are useful for modeling ROMs (e.g., for control logic) or simply for loading programs into the main memory of your Beta. One caveat: if the memory has a write port and sees a rising clock edge with its write enable not equal to 0 and with one or more of the address bits undefined (i.e., with a value of "X"), the entire contents of the memory will also become undefined. So you should **make sure that the write enable for a write port is set to 0 by your reset logic** before the first clock edge, or else your initialization will be for naught.

The following options can be used to specify the electrical and timing parameters for the memory. For this lab, these should not be specified and the default values used.

`tcd=`*`seconds`*
the contamination delay in seconds. Default value = 20ps.

`tpd=`*`seconds`*
the propagation delay in seconds. This is how long it takes for changes in the address or output enable terminals to be reflected in the values driven by the data terminals. Default value is determined from the number of locations:

Number of locations	t_{PD}	Inferred type
nlocations ≤ 128	2ns	Register file
128 < nlocations ≤ 1024	4ns	Static ram
nlocations > 1024	40ns	Dynamic ram

`tr=`*`seconds_per_farad`*
the output rise time in seconds per farad of output load. Default value is 1000, i.e., 1 ns/pf.

`tf=`*`seconds_per_farad`*
the output fall time in seconds per farad of output load. Default value is 500, i.e., 0.5 ns/pf.

`cin=`*`farads`*
input terminal capacitance in farads. Default value = 0.05pf.

`cout=`*`farads`*
output terminal capacitance in farads. Default value = 0pf (additional t_{PD} due to output terminal loading is already included in default t_{PD}).

The size of a memory is determined by the sum of the sizes of the various memory building blocks shown in the following table:

Component	*Size (μ^2)*	*Notes*

Storage cells	nbits * cellsize	nbits = nlocs * width
		cellsize = nports (for ROMs and DRAMS)
		cellsize = nports + 5 (for SRAMS)

Address buffers	nports * naddr * 20	nports = total number of memory ports
Address decoders	nports * (naddr+3)/4 * 4	Assuming 4-input ANDs
Tristate drivers	nreads * width * 30	nreads = number of read ports
write-data drivers	nwrites * width * 20	nwrites = number of write ports

Appendix B – ROM Control Logic

ROM control logic uses a ROM to implement all the signals needed where the ROM is defined using the memory modules discussed above. The module definition includes `contents` where you can describe how each control signal acts for a given address. Implementing this would look something like:

```
Xctl vdd 0 0 id[31:26]          // one read port
+ pcsel[2:0] wasel asel ra2sel bsel alufn[4:0] wdsel[1:0] werf moe xwr
+ $memory width=17 nlocations=64 contents=(
+  0b00000000000000000         // opcode=0b000000
+  0b00000000000000000         // opcode=0b000001
+ …
+ )
```