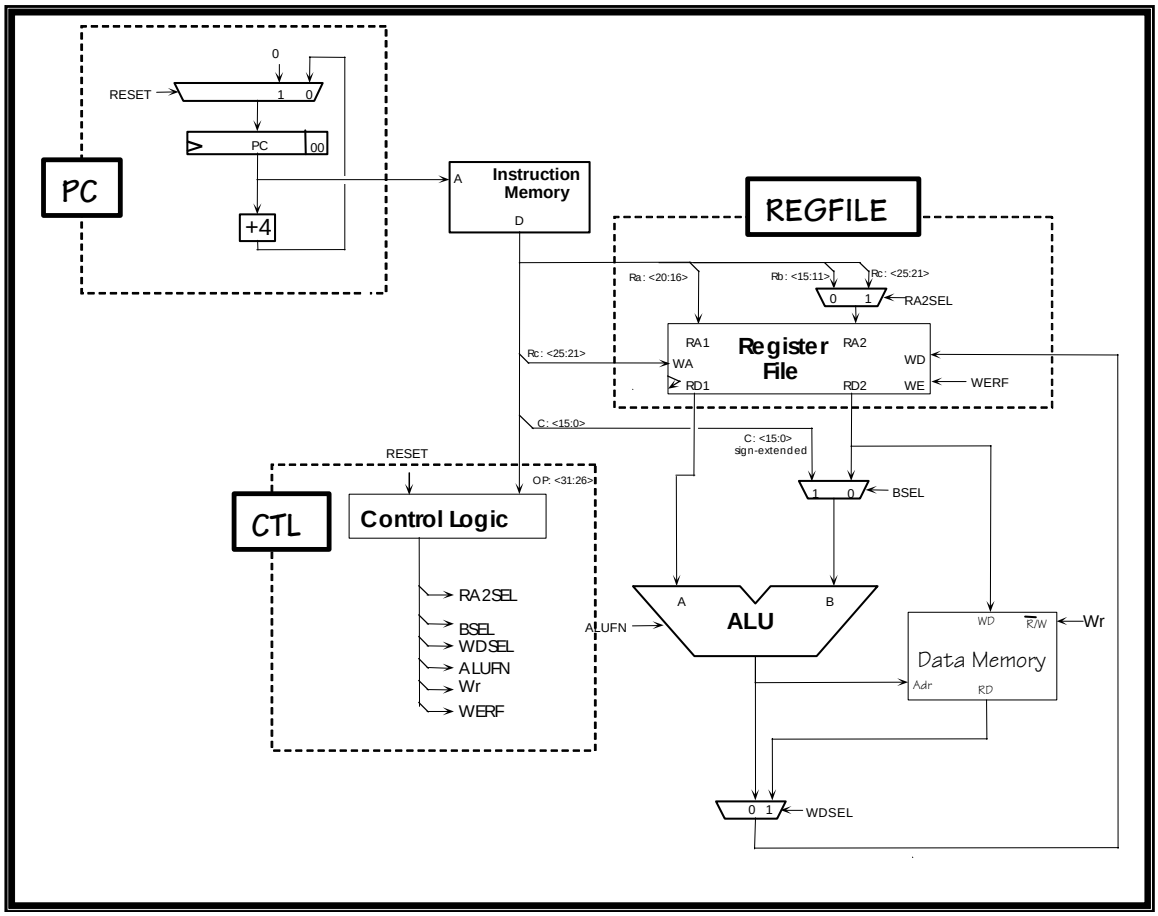
M A S S A C H U S E T T S I N S T I T U T E O F T E C H N O L O G Y
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.004 Computation Structures
Lab #6

The goal of this lab is to complete your Beta design to the point where it can execute a "basic block" of instructions, i.e., a sequence of instructions that does not contain any branches or jumps. The following diagram of a simplified Beta that can execute basic blocks shows what needs to be done:



It's probably best to tackle the design in stages. Here are some design notes keyed to the block diagram shown above.

PC

The 32-bit multiplexer selects the value to be loaded into the PC at next rising edge of the clock. Eventually the mux will have inputs used for implementing branches, jumps, exceptions, etc. but for now use a two-input 32-bit mux that selects 0x00000000 when the RESET signal is asserted, and the output of the PC+4 logic when RESET is not asserted. We will use the RESET signal to force the PC to zero during the first clock period of the simulation.

The PC is a separate 32-bit register that can be built using the `dreg` component from the parts library. You should include hardware for the bottom two bits of the PC even though they are always 0; this will make debugging traces easier to interpret.

Conceptually, the increment-by-4 circuit is just a 32-bit adder with one input wired to the constant 4. You can implement it this way, but it is possible to build a much smaller circuit if you design an adder optimized knowing that one of its inputs is 0x00000004.

We've created a test jig to test your PC circuitry. Your netlist should incorporate the following three .include statements

```
.include "/mit/6.004/jsim/nominal.jsim"
.include "/mit/6.004/jsim/stdcell.jsim"
.include "/mit/6.004/jsim/lab6pc.jsim"
```

and the following subcircuit definition (you can of course define other subcircuits as well)

```
.subckt pc clk reset ia[31:0]
… your mux/register/+4 circuit here …
.ends
```

To use the test jig, make sure your design file contains a definition for an "pc" subcircuit as shown above. Then do a gate-level simulation; a waveform window showing the pc inputs and outputs should appear. Next click the checkoff button (the green checkmark) in the toolbar. JSim will check your circuit's results against a list of expected values and report any discrepancies it finds. Using this test jig file, nothing will be sent to the on-line server – it's provided to help test your design as you go.

REGFILE

The register file is a 3-port memory. Here's a template netlist for specifying the 3-port register file:

```
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=31
```

A more complete description of how to use the $memory JSim component can be found at the end of this writeup.

Note that the memory component doesn't know that location 31 of the register file should always read as zero, so you'll have to add additional logic around the memory that makes this happen. You can use muxes or ANDs to force the register data for each read port to "0" when the port address = 0b11111 (i.e., R31). And you'll need a mux to implement the RA2SEL circuitry.

We've created a test jig to test your register file circuitry. Your netlist should incorporate the following three .include statements

```
.include "/mit/6.004/jsim/nominal.jsim"
.include "/mit/6.004/jsim/stdcell.jsim"
.include "/mit/6.004/jsim/lab6regfile.jsim"
```

and the following subcircuit definition (you can of course define other subcircuits as well)

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

CTL

The control logic should be tailored to generate the control signals *your* logic requires, which may differ from what's shown in the diagram above. Note that a ROM can be built by specifying a memory with just one read port; the ROM contents are set up using the `contents` keyword in the netlist description of the memory. For example, the netlist for a ROM that uses the opcode field of the instruction to lookup the values for 12 control signals might look like:

```
Xctl vdd 0 0 id[31:26]    // one read port
+ ra2sel bsel alufn[5:0] wdsel werf moe xwr
+ $memory width=12 nlocations=64 contents=(
+  0b000000000000         // opcode=0b000000
+  0b000000000000         // opcode=0b000001
+ …
+ )
```

Most of the signals can connected directly to the appropriate logic, e.g., alufn[5:0] can connect directly to the ALUFN inputs of your ALU.

We do need to be careful with the write enable signal for main memory (WR) which needs to be valid even before the first instruction is fetched from memory. So you should include some **additional logic that forces WR to 0 when RESET=1** – the signal XWR from the ROM needs to combined appropriately with RESET to form WR. MOE is another memory control signal; see the next section for more information.

We'll be adding more bits to the control ROM when the branch logic is added in the next lab. For this lab, your design should implement the following instructions:

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

Eventually unimplemented instructions will cause an exception, but for now turn them into NOPs by making sure WERF is set to 0 (preventing any value from being written into a destination register).

If you've followed the scheme outlined above, we've created a test jig to test your control circuitry. Your netlist should incorporate the following three .include statements

```
.include "/mit/6.004/jsim/nominal.jsim"
.include "/mit/6.004/jsim/stdcell.jsim"
.include "/mit/6.004/jsim/lab6ctl.jsim"
```

and the following subcircuit definition (you can of course define other subcircuits as well)

```
.subckt ctl reset id[31:26] ra2sel bsel alufn[5:0] wdsel werf moe mwe
… your control circuit here …
.ends
```

Finishing the design

Memories: The instruction and data memories will be supplied by lab6checkoff.jsim; you just need to supply the necessary address, data and control signals:

instruction address (ia[31:0], output). Address of next instruction to be executed.

instruction data (id[31:0], input). After the appropriate propagation delay, the memory will drive these signals with the contents of the memory location specified by ia[31:0].

data memory address (ma[31:0], output). Address of data memory location to be read or written.

memory output enable (moe, output). Set to 1 when the Beta wants the memory to read the memory location specified by ma[31:0].

memory read data (mrd[31:0], input). If moe is 1, the memory will drive these signals with the contents of the memory location specified by ma[31:0].

memory write enable (wr, output). Set to 1 when the Beta wants to store into the memory location specified by ma[31:0] at the end of the current cycle. **NOTE: this signal should always have a valid logic value at the rising edge of CLK otherwise the contents of the memory will be erased. You'll need to take care in designing the logic that generates this signal – see above for details.**

memory write data (mwd[31:0], output). If wr is 1, this is the data that will be written into memory at the end of the current cycle.

Your Beta design should include circuitry to generate all the signals labeled as "output". Signals labeled as "input" will be driven by memory.

BSEL mux: The low-order 16 bits of the instruction need to be sign-extended to 32 bits. Sign-extension is easy in hardware! Just connect inst[15:0] to the low-order sixteen D1 inputs of the mux and inst15 to each of the high-order sixteen D1 inputs.

WDSEL mux: In the final design the 32-bit WDSEL mutiplexer will select the data to be written into the register file from one of three possible sources. For this lab, there are only two choices: the output of the ALU and read data from main memory (i.e., the data on mrd[31:0]).

When you've completed your design, you can use lab6checkoff.jsim to test your circuit and complete the checkoff. Your netlist should incorporate the following three .include statements

```
.include "/mit/6.004/jsim/nominal.jsim"
.include "/mit/6.004/jsim/stdcell.jsim"
```

```
.include "/mit/6.004/jsim/lab6checkoff.jsim"
```

and the following subcircuit definition (you can of course define other subcircuits as well)

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

Your design will be tested at a cycle time of 100ns. The reset signal is asserted for the first clock edge and then deasserted to start the program running. This implementation of the Beta subcircuit has the following terminals:

clk	input	clock (from test circuitry)
reset	input	reset (from test circuitry)
ia[31:0]	outputs	instruction address (from PC register)
id[31:0]	inputs	instruction data (from test circuitry)
ma[31:0]	outputs	memory data address (from ALU)
moe	output	memory read data output enable (from control logic)
mrd[31:0]	inputs	memory read data (from test circuitry)
wr	output	memory write enable (from control logic)
mwd[31:0]	outputs	memory write data (from register file)

Lab6checkoff.jsim uses the following netlist to create the test circuitry:

```
// create an instance of the Beta to be tested
Xbeta clk reset ia[31:0] id[31:0] ma[31:0]
+ moe mrd[31:0] wr mwd[31:0] beta

// memory is word-addressed and has 1024 locations
// so only use address bits [11:2].
Xmem
+ vdd 0 0 ia[11:2] id[31:0]     // port 1: instructions (read)
+ moe 0 0 ma[11:2] mrd[31:0]    // port 2: memory data (read)
+ 0 clk wr ma[11:2] mwd[31:0]   // port 3: memory data (write)
+ $memory width=32 nlocations=1024 contents=(
+ … binary representation of /mit/6.004/bsim/lab6.uasm …
+ )

// clock has 100ns cycle time, starts as 1 so first clock
// edge happens 100ns into the simulation
Vclk clk 0 pulse(3.3,0,49.9ns,.1ns,.1ns,49.9ns,100ns)

// reset starts as 1, set to 0 just after first clock edge
Vreset reset 0 pwl(0ns 3.3v, 101ns 3.3v, 101.1ns 0v)
```

Lab6checkoff.jsim checks out your design by attempting to run a test program and verifying that your Beta outputs the correct values on its outputs every cycle. The source for the test program can be found at

/mit/6.004/bsim/lab6.uasm

The test program computes a “magic number” using a sequence of operations that tests the various pieces of your design. The program writes out a series of results into memory location 0x3FC; look at lab6.uasm to see what values are written on which cycles.

Lab6checkoff.jsim will verify that the instruction address (ia[31:0]), memory address (ma[31:0]), memory write data (mwd[31:0]) and the memory control signals (moe, wr) have the correct values each cycle. The check is made just before the rising clock edge, i.e., after the current instruction has been fetched and executed, but just before the result is written into the register file. Note that ma[31:0] is the output of the ALU, so these checks can verify that all instructions are working correctly. If you get a verification error, check the instruction that has just finished executing at the time reported in the error message – your Beta has executed that instruction incorrectly for some reason.

Almost nobody's Beta design executes the test program correctly the first time! It will take some effort to debug your design, but stick with it. When your circuitry works, congratulations! We'll be able finish the Beta design in the next lab.

Optional: If your design contains any registers or memories, JSim will report the "minimum observed setup time" at the end of each simulation run. At each rising clock edge, JSim determines the setup time for each data input to a register or memory (i.e., how long the data inputs were valid before the rising clock edge). JSim remembers the smallest setup time it finds, along with the simulated time it made the observation and the device involved. So, for example, JSim might report

> min observed setup = 85.235ns @ time=5.2us (device = xbeta.xregfile)

For a Beta design, the reported device is almost always the register file—this makes sense since the last signals to settle should be WDATA[31:0], the data inputs to the register file.

Since the tests are run with a clock period of 100ns, this tells us that we could have reduced the clock period to (100 – 85.235 + $t_{MEM\ SETUP}$)ns and still expect the test program to run correctly. We can look at the waveform plots to determine what instruction had just finished executing at time 5.2us – that's the instruction whose execution we'd have to speed up in order to reduce the cycle time of our Beta. Typically the worst-case execution time comes from either CMPxx or LD instructions (why?).

Using this technique, investigate where the critical path(s) are in your Beta design and work to make them as short as possible. To get the fastest possible cycle time you'll probably need to implement some of your control signals (e.g., RA2SEL) using logic gates rather than a ROM. Given that we might have to make three memory accesses in a single cycle (instruction fetch + register file access + data memory access = 10ns total assuming a 1024-location main memory), we won't be able to do better than 100Mhz clock rates unless we pipeline our implementation.

Using the JSim memory component

We'll be using a new component this week: 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

```
Xid 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 "lab6.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="/mit/6.004/bsim/lab6.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