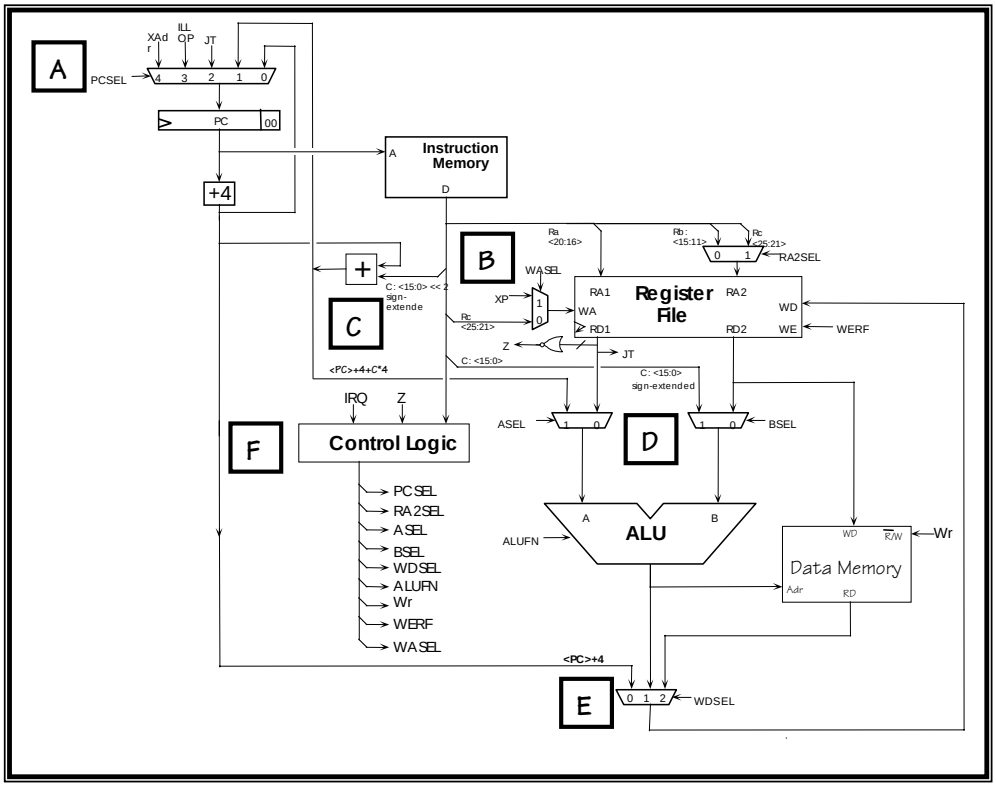
M A S S A C H U S E T T S I N S T I T U T E O F T E C H N O L O G Y
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.004 Computation Structures
Lab #7

In this lab you'll complete the implementation of your Beta. We'll take the block diagram from Lab #6 and add the remaining logic:



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

(A) Since the parts library doesn't have any 5-input multiplexers, you'll have to construct the logic that selects the next PC using other components and adjust the control logic accordingly. Remember to add a way to set the PC to zero on reset (see part F). "XAdr" and "ILL OP" in the block diagram represent constant addresses used when the Beta services an interrupt (triggered by IRQ) or executes an instruction with an illegal or unimplemented opcode. For this assignment assume that XAdr=8 and ILLOP=4 and we'll make sure the first three locations of main memory contain BR instructions that branch to code which handle reset, illegal instruction traps and interrupts respectively. In other words, the first three locations of main memory contain:

```
Mem[0x00000000] = BR(reset_handler)
Mem[0x00000004] = BR(illop_handler)
Mem[0x00000008] = BR(interrupt_handler)
```

Note on supervisor bit: The high-order bit of the PC is dedicated as the "Supervisor" bit (see section 6.3 of the Beta Documentation). Instruction fetch and the LDR instruction ignore this bit, treating it as if it were zero. The JMP instruction is allowed to clear the Supervisor bit or leave it unchanged, but cannot set it, and *no other instructions may have any effect on it*. Only reset, exceptions and interrupts cause the Supervisor bit to become set. This has the following implications for your Beta design:

1. 0x80000000, 0x80000004 and 0x80000008 are loaded into the PC during reset, exceptions and interrupts respectively. This is the only way that the supervisor bit gets set. Note that after reset the Beta starts execution in supervisor mode.

2. Bit 31 of the PC+4 and branch-offset inputs to the PCSEL mux should be connected to PC31, i.e., the value of the supervisor bit doesn't change when executing most instructions.

3. You'll have to add logic to bit 31 of the JT input to the PCSEL mux to ensure that JMP instruction can only clear or leave the supervisor bit unchanged. Here's a table showing the new value of the supervisor bit after a JMP as function of JT31 and the current value of the supervisor bit (PC31):

old PC31	JT31	new PC31
0	--	0
1	0	0
1	1	1

4. Bit 31 of the branch-offset input to the ASEL mux should be set to 0 – the supervisor bit is ignored when doing address arithmetic for the LDR instruction.

5. Bit 31 of the PC+4 input to the WDSEL mux should connect to PC31, saving the current value of the supervisor whenever the value of the PC is saved by a branch instruction or trap.

(B) The 5-bit 2-to-1 WASEL multiplexer determines the write address for the register file.

(C) The branch-offset adder adds PC+4 to the 16-bit offset encoded in the instruction. The offset is sign-extended to 32-bits and multiplied by 4 in preparation for the addition. Both the sign extension and shift operations can be done with appropriate wiring—no gates required!

(D) The ASEL multiplexer and the Z logic are added to the output of RA1/RD1 port of the register file. This port is also wired directly to the JT inputs of the PCSEL multiplexer (remember to force the low-order two bits to zero and to add supervisor bit logic to bit 31!).

(E) The WDSEL multiplexer is expanded to 3 inputs so that PC+4 can be stored in the register file during execution of BEQ, BNE, JMP and exceptions. Note that the supervisor bit (PC31) is saved away along with the rest of the PC.

(F) 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. The new logic we've added

requires the generation of several new control signals:

PCSEL[2:0] – determines the next value of PC
ASEL – control signal for ASEL mux on A input of ALU (for LDR instruction)
WDSEL[1:0] – expanded to include the third (PC+4) input
WASEL – selects XP as the destination register during exceptions

This version of the Beta should implement the LDR, JMP, BNE, and BEQ instructions; when implementing the latter two instructions the PCSEL signals are determined using the "Z" signal (Z = 1 when the value of Reg[Ra] is zero). If you are using a ROM-based implementation, you can make Z an additional address input to the ROM (doubling its size). A more economical implementation might use external logic to modify the value of the PCSEL signals.

An interrupt-request (IRQ) input has been added to the Beta. When this signal is 1 and the Beta is in "user mode" (PC31 is zero), an interrupt should occur. **Asserting IRQ should have no effect when in "supervisor mode" (PC31 is one).** You should add logic that causes the Beta to abort the current instruction and save the current PC+4 in register XP and to set the PC to 0x80000008. In other words, an interrupt forces the following:

1. PCSEL to 4 (select 0x80000008 as the next PC)
2. WASEL to 1 (select XP as the register file write address)
3. WERF to 1 (write into the register file)
4. WDSEL to 0 (select PC+4 as the data to be written into the register file)
5. WR to 0 (this ensures that if the interrupted instruction was a ST that it doesn't get to write into main memory).

Note that you'll also want to add logic to reset the Beta; at the very least when reset is asserted you'll need to force the PC to 0x80000000 and ensure that WR is 0 (to prevent your initialized main memory from being overwritten).

Your design should implement all the instructions listed in the Beta Documentation, except for MUL, MULC, DIV and DIVC, which are optional. Unimplemented instructions should cause an exception as outlined in part (A) above.

When you've completed your design, you can use lab7checkoff.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/lab7checkoff.jsim"
```

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

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

Note the addition of the IRQ (interrupt request) input. 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)
irq	input	interrupt request (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)

Lab7checkoff.jsim uses the following netlist to create the test circuitry:

```
// create an instance of the Beta to be tested
Xbeta clk reset irq 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].  moe

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/lab7.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)

// interrupt request is asserted twice.  The first time (during
// cycle 10) should be ignored because the Beta is in supervisor
// mode, the second time (during cycle 273) should cause an
// interrupt.
Virq irq 0 pwl(0ns 0v, 1001ns 0v,
+ 1001.1ns 3.3v,
+ 1101ns 3.3v,
+ 1101.1ns 0v,
+ 27301ns 0v,
+ 27301.1ns 3.3v,
+ 27401ns 3.3v,
+ 27401.1ns 0v
+)
```

Lab7checkoff.jsim checks out your design by attempting to run a test program and verifying that your Beta outputs the correct value on its outputs every cycle. The source for the test program can be found at

/mit/6.004/bsim/lab7.uasm

The checkoff program attempts to exercise all the features of the Beta architecture. If this program completes successfully, it enters a two-instruction loop at locations 0x3C4 and 0x3C8. It reaches 0x3C4 for the first time on cycle 277.

Lab7checkoff.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 – the Beta has executed that instruction incorrectly for some reason.

Almost nobody's design executes the checkoff program correctly the first time! To understand what went wrong, you'll need to retrieve the error code and compare it with the table given at the beginning of lab7.uasm. The table will indicate at what label the program detected an error; for example if the error code is 0x288, then the checkoff program detected an error in the code just before label "bool1" in the program. Looking through lab7.uasm, you can locate the "bool1" label and see what results the program expected. Now look at the waveforms of your Beta executing the same code and you can usually track down the error in your design.

It will take some effort to debug your design, but stick with it! If you're stuck, get help from your fellow students or the course staff. When it works, congratulate yourself: the design of a complete CPU at the gate-level is a significant accomplishment. Of course, now the fun is just beginning—there are undoubtedly many ways you can make improvements, both large and small. Good luck!