CS 3220
A Tour of a Five-Stage Processor
The Stages

• Classical 5-stage pipeline (like in CS 2200)
  – Fetch
  – Decode
  – ALU
  – Memory
  – Write Register

• Signal Naming Issues
  – Many signals go through multiple stages
  – Need to give them separate names
  – We’ll use sig_F, sig_D, sig_A, sig_M
  – E.g. wregno_M is wregno in the M(EM) stage
What Each Stage Does

- Obvious stuff – what the name of the stage says
- Additional stuff
  - Fetch – also do PC prediction
  - Decode – also read register, start PCcorr computation
  - ALU – also do most of the PCcorr computation
    - Possible to complete PCcorr here but makes cycle longer
  - MEM – lots of additional stuff here
    - Read/Write memory-mapped device registers
    - Generate PCcorr and mispred signals
    - Take interrupt if needed, update system registers if int or RETI
    - Read/write system registers for RSR/WSR
  - Write Reg – also update bpred table
Tour Itinerary

• First we’ll look at what happens in the “datapath”
  – Things that determine memory and register values
  – Also includes forwarding and stall signals
• Then we’ll look at “control flow” issues
  – Things that determine what gets fetched
• Finally, we’ll look at interrupt-related stuff
  – Including how to check and update S* registers
• Almost trivial – PC to imemaddr, get inst_F
The Datapath – FtoD Flip-Flops

• Trivial, except if flush or stall

```verilog
reg [(DBITS-1):0] inst_D;
reg flushed_D;
always @(posedge clk) if(lock) begin
    if(flush_F)
        {inst_D,flushed_D}<={IW_NOP,1'b1};
    else if(!stall_D)
        {inst_D,flushed_D}<={inst_F,1'b0};
end
```

• What is the purpose of flushed_D signal?
  – Indicates to later stage if this is a real inst or a bubble
  – Why? Because it may come handy 😊
The Datapath – Decode

- Take inst_D, generate all sorts of signals
  - Give nice names to parts of inst_D first

  ```
  parameter REGNOBITS=5;
  parameter IMMBITS=16;
  wire [5:0]          opcode1_D=inst_D[31:26];
  wire [(REGNOBITS-1):0] rx_D    =inst_D[25:21];
  wire [(REGNOBITS-1):0] ry_D    =inst_D[20:16];
  wire [(REGNOBITS-1):0] rz_D    =inst_D[15:11];
  wire [5:0]          opcode2_D=inst_D[ 5: 0];
  wire [(IMMBITS-1):0] imm_D    =inst_D[(IMMBITS-1): 0];
  ```

- This is not really needed
  - Could use inst_D[31:26] instead of opcode1_D
  - But opcode1_D is more descriptive
The Datapath – Decode

- Generate (almost) all control signals
  – Not just those we need in the D stage

```vhdl
case(opcode1_D)  
  OP1_ALU: begin  
    case(opcode2_D)  
      ALU_OP2_ADD:  
        {rdreg1_D, rdreg2_D, aluimm_D, selarithmetic_D, arithsub_D, wregno_D, wrreg_D} =  
        {1'b1, 1'b1, 1'b0, 1'b1, 1'b0, rdst_D, 1'b1};  
        ...  
      ALU_OP2_NXOR:  
        {rdreg1_D, rdreg2_D, aluimm_D, sellogic_D, logicop_D, logicneg_D, wregno_D, wrreg_D} =  
        {1'b1, 1'b1, 1'b0, 1'b1, LOGIC_XOR, 1'b1, rdst_D, 1'b1};  
      default:  
        iinst_D = 1'b1;  
    endcase  
  end
end

OP1_ADDI:  
  {rdreg1_D, selarithmetic_D, arithsub_D, aluimm_D, wregno_D, wrreg_D} =  
  {1'b1, 1'b1, 1'b0, 1'b1, rsrv2_D, 1'b1};  

OP1_BEQ:  
  {rdreg1_D, rdreg2_D, btest_D, bcond_D} =  
  {1'b1, 1'b1, 1'b1, BCOND_EQ};
```
The Datapath – Decode

- And read registers, of course

```verilog
wire [2:0] rregno1_D=rx_D, rregno2_D=rt_D;
wire [(DBITS-1):0] regout1_D,regout2_D;
RegFile #(.DBITS(DBITS),.ABITS(REGNOBITS),.MFILE("Regs.mif"))
regFile(.RADDR1(rregno1_D),.DOUT1(regout1_D),
    .RADDR2(rregno2_D),.DOUT2(regout2_D),
    .WADDR(wregno_W), .DIN(result_W),
    .WE(wrreg_W),.CLK(clk));
```
• Do forwarding and generate stall signals!

```verilog
wire forw1A_D=wrreg_A&& (rregno1_D==wregno_A);
wire forw2A_D=wrreg_A&& (rregno2_D==wregno_A);
wire stallA_D= ((forw1A_D&& rdreg1_D) || (forw2A_D&& rdreg2_D)) &&
                   (!gotresult_A);
wire forw1M_D=wrreg_M&& (rregno1_D==wregno_M);
wire forw2M_D=wrreg_M&& (rregno2_D==wregno_M);
wire forw1W_D=wrreg_W&& (rregno1_D==wregno_W);
wire forw2W_D=wrreg_W&& (rregno2_D==wregno_W);
wire [(DBITS-1):0] regval1_D=forw1A_D?result_A:
                              forw1M_D?result_M:
                              forw1W_D?result_W:
                              regout1_D;

wire [(DBITS-1):0] regval2_D=forw2A_D?result_A:
                              forw2M_D?result_M:
                              forw2W_D?result_W:
                              regout2_D;

wire stall_D=stallA_D;
wire stall_F=stall_D;
```
always @(posedge clk) begin
  {regval1_A, regval2_A, imm_A, aluimm_A} <= 
  {regval1_D, regval2_D, imm_D, aluimm_D};
  {logicop_A, logicneg_A, arithsub_A, complt_A, compeq_A} <= 
  {logicop_D, logicneg_D, arithsub_D, complt_D, compeq_D};
  {selpcplus_A, sellogic_A, selarith_A, selcomp_A, selmem_A, selsreg_A} <= 
  {selpcplus_D, sellogic_D, selarith_D, selcomp_D, selmem_D, selsreg_D};
  {wrmem_A, rregno1_A, wsreg_A, reti_A, wrreg_A, wregno_A} <= 
  {wrmem_D, rregno1_D, wsreg_D, reti_D, wrreg_D, wregno_D};
  {iinst_A, sinst_A, flushed_A} <= 
  {iinst_D, sinst_D, flushed_D};
if(flush_D || stall_D) 
  {wrmem_A, wsreg_A, reti_A, wrreg_A, iinst_A, sinst_A, flushed_A} <= 
  {1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b1};
end
The Datapath – Fast ALU

• Determine ALU inputs
  ```
  wire [(DBITS-1):0] aluin1=regval1_A;
  wire [(DBITS-1):0] aluin2=aluimm_A?dimm_A:regval2_A;
  ```

• Generate result for each operation
  ```
  wire [(DBITS-1):0] resarith_A=alin1+(alin2^{DBITS{arithsub_A}})+arithsub_A;
  wire rescomp_A=((alin1<alin2)&&complt_A)||(alin1==alin2)&&compeq_A);
  wire [(DBITS-1):0] vallogic_A=
    (logicop_A==LOGIC_AND)?(alin1&alin2):
    (logicop_A==LOGIC_OR )?(alin1|alin2):
    (logicop_A==LOGIC_XOR)?(alin1^alin2):
    {DBITS{1'bX}};
  wire [(DBITS-1):0] reslogic_A={DBITS{logicneg_A}}^vallogic_A;
  ```

• Select final ALU result
  – Use sel*_A signals

• Memory address? It’s in resarith_A!
The Datapath – A2M Flip-Flops

- Fewer signals (ALU control not needed here)
- No stalls here
  - Stalls are needed to delay instruction until forwarding can provide correct register values

always @(posedge clk) begin
  {regval1_M, regval2_M, rregno1_M, wsreg_M, reti_M, wregno_M, wrreg_M}<=
  {regval1_A, regval2_A, rregno1_A, wsreg_A, reti_A, wregno_A, wrreg_A};
  {dmemaddr_M, wrmem_M, restmp_M, gotrestmp_M}<=
  {dmemaddr_A, wrmem_A, result_A, gotresult_A};
  {selmem_M, selsreg_M}<= {selmem_A, selsreg_A};
  {iinst_M, sinst_M, flushed_M}<=
  {iinst_A, sinst_A, flushed_A};
  if (flush_A) begin
    {wrmem_M, wsreg_M, reti_M, wrreg_M, iinst_M, sinst_M, flushed_M}<=
    {1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b1};
  end
end
The Datapath – Memory

- Feed `dmemaddr_M` and `wrmem_M` to memory and I/O devices
  - They read/write the actual memory and I/O registers
  - Reads put values on `dbus`, which feeds `result_M`
  - Code for this given out with Project 4 writeup
• Very simple FFs – no stalls, no flushes
  – If an inst got to M stage, it is “safe”
    
    reg wrreg_W=1’b0;  
    reg [(REGNOBITS-1):0] wregno_W;  
    reg [(DBITS-1):0] result_W;  
    always @(posedge clk) if(!init) begin
            {wrreg_W,wregno_W,result_W}<=  
             {wrreg_M,wregno_M,result_M};  
    end

• No “real” code for the W stage
  – Just feed wrreg_W, wregno_W, and result_M signals to the register file
• Update PC every cycle

```verilog
wire [(DBITS-1):0] PCpred_F;
always @(posedge clk) if(lock) begin
    if(init)
        PC<=16'h200;
    else if(mispred_M)
        PC<=pcgood_M;
    else if(!stall_F)
        PC<=pcpred_F; // From bptable
end
```

• Compute Pcpplus (PC+4)

```verilog
wire [(DBITS-1):0] pcplus_F=PC+32'd4;
```

• Branch predictor
  – We’ve already seen the code for this
Control Flow – FtoD Flip-Flops

• Carry PC, PCplus, PCpred to next stage

```verilog
always @(posedge clk) begin
  if(flush_F)
    {PC_D,pcplus_D,pcpred_D}<=
    {32'b0,pcgood_B,pcgood_B};
  else if(!stall_D)
    {PC_D,pcplus_D,pcpred_D}<=
    {PC,pcplus_F,pcpred_F};
end
```

```verilog
always @(posedge clk) if(lock) begin
  if(flush_F)
    {inst_D,flushed_D}<=
    {IW_NOP,1'b1};
  else if(!stall_D)
    {inst_D,flushed_D}<=
    {inst_F,1'b0};
end
```

• Note how flush and stall mirrors datapath FtoD
  – It’s really one big FtoD set of FFs!
  – We just separate them in our code for clarity
• Compute btarg, jtarg is just regval1_D

wire [(DBITS-1):0] bimm_D={((DBITS-IMMBITS-1){imm_D[IMMBITS-1]})},imm_D,1'b0;
wire [(DBITS-1):0] btarg_D=pcplus_D+bimm_D;
wire [(DBITS-1):0] jtarg_D=regval1_D;

• What else do we need to compute pcgood?

• Why not compute pcgood here?
The pcgood saga continues!
  – Find if regval1_A==regval2_A
  • Now that we have these values!
  – Use this and bcond_A to determine btake_A

wire iseq_A=(regval1_A==regval2_A);
parameter
  BCOND_EQ=1'b0,
  BCOND_NE=1'b1;
wire btake_A=btest_A&&((bcond_A==BCOND_EQ)?(iseq_A):
  (bcond_A==BCOND_NE)?(!iseq_A):
  1'bX);

Now we can determine pcgood!
  – But do we want to?
And then find mispred as (pcgood!=pcpred)
  – And then feed it to the PC-selection logic in F stage
• Take btarg_M, jtarg_M, btake_M, etc. and finally determine pcgood

• Also compute mispred_M
  – Can compute it as (pcgood_M!=pcpred_M)
    • Takes some time AFTER the pccorr is computed
    • And then we need to get mispred_M across wires to PC selection
  – There is a MUCH better way to generate mispred_M
    • Think: can I do some comparisons in earlier stages (not with pcgood) so that I can determine mispred_M quickly in the M stage
    • This would give mispred_M more time to get to the F stage
• But we already did pcgood and mispred stuff!
  – What is left to do?
• Update the branch predictor table!

```verilog
reg [(DBITS-1):0] PC_W, pcgood_W;
always @(posedge clk) begin
  {PC_W, pcgood_W} <= {PC_M, pcgood_M};
end

wire [(BPABITS-1):0] bpUpdInd = PC_W[BPABITS:1];
always @(posedge clk) if (PC_W != 16'b0) begin
  bptable[bpUpdInd] <= pcgood_W;
end
```

• Why don’t we have “if(init) …” stuff in this code?
System/Interrupt Support

• It’s all in M stage
  – S* registers read/written in M stage for RSR/WSR
    • Eliminates forwarding of S* register values from WSR to RSR
    • RSR and LW use the same forwarding to other insts
  – So S* reads and writes for int/RETI are also in M stage
    • Eliminates forwarding of S* values between int/RETI and WSR/RSR
  – And int/RETI will keep insts in M stage and flush F,D,A
    • Same flush as for branch misprediction (assuming Pccorr done in M)

• If we had pcgood computation finish in A stage?
  – Then we would have all of this in the A stage
  – Don’t want to have two different kinds of flushes!
    • Doable, but we do want to finish Project 5 this semester 😊
• Declare the registers!

\[
\begin{align*}
\text{reg IE, OIE, CM, OM;} \\
\text{reg [(DBITS-1):0] SIH, SRA, SII, SR0, SR1;}
\end{align*}
\]

• Reading them on RSR?

\[
\begin{align*}
\text{reg [(DBITS-1):0] sregout\_M;} \\
\text{always @(rregno1\_M or IE or OIE or CM or OM or SIH or SRA or ...)} \\
\quad \text{case}(rregno1\_M) \\
\quad \text{SREG\_SCS: sregout\_M} = \{(DBITS-4)\{1'b0\}, OM, CM, OIE, IE\}; \\
\quad \text{SREG\_SIH: sregout\_M} = \text{SIH}; \\
\quad \text{SREG\_SRA: sregout\_M} = \text{SRA}; \\
\quad \text{SREG\_SII: sregout\_M} = \text{SII}; \\
\quad \text{SREG\_SR0: sregout\_M} = \text{SR0}; \\
\quad \text{SREG\_SR1: sregout\_M} = \text{SR1}; \\
\quad \text{default: sregout\_M} = 16'hFAFA; \\
\end{align*}
\]

\[
\begin{align*}
\text{assign rbus=selsreg\_M?sregout\_M:DBITS\{1'bz\};}
\end{align*}
\]

// Can we have this to get ALU result to go to register?
\[
\begin{align*}
\text{assign rbus}=(!selmem\_M):restmp\_M:DBITS\{1'bz\};
\end{align*}
\]
Some of these registers are updated by

- WSR, e.g. WSR PCS, A0
- Taking an illegal instruction exception
  - Illegal inst in A stage, if not flushed by what we have in M stage
- Taking interrupt from I/O devices
  - Interrupt request is active and IE is 1

All three may need to be considered each cycle

- WSR changes IE from 0 to 1 and CM from 1 to 0
- Next instruction is RSR (illegal if CM is 0)
- There is an interrupt request pending
Example: Updating IE

- What can change IE?
  - Instruction in M stage changes IE
    - WSR
    - RETI
  - Interrupt taken (change IE to 0)

- Which value of IE do we use?
  - Lets try the value from the beginning of the cycle
Update in Several Steps

• Current M-stage inst vs. illegal A-stage inst
  – Current instruction changes what is legal for next inst, so current instruction updates S* first

• Current M-stage inst vs. interrupt
  – Current instruction may disable interrupts, interrupt should not appear right after it

• Illegal inst vs. interrupt
  – Each would change IE to 0, if interrupt goes first it gets higher priority than illegal inst

• OK, so we have to consider current M-stage inst first, then illegal A-stage inst, then interrupts
• Do what RETI and WSR need to do
  – Produce temporary values (don’t write to regs yet!)

```verilog
reg OIEinst_M,IEinst_M;
reg [(DBITS-1):0] IHAinst_M, IRAinst_M, IDNinst_M;
always @(wsreg_M or wregno_M or regval1_M or creti_M or OM or ...) begin
  {OIEinst_M,IEinst_M}=
  {OIE    ,IE   };
  {IHAinst_M,IRAinst_M,IRAinst_M}=
  {IHA    ,IRA   ,IRA   };
  if(creti_M)
    IEinst_M=OIE;
  else if(wsreg_M) case(wregno_M)
    SREG_PCS: {OIEinst_M,IEinst_M}=regval1_M[1:0];
    SREG_IHA: IHAinst_M=regval1_M;
    SREG_IRA: IRAinst_M=regval1_M;
    ...
  endcase
end
```
Effects of Device Interrupt Requests

- Request causes an interrupt
  - But only if IE still 1!
  - And be careful about placing a correct address in SRA

```verilog
reg OIEdev_M, IEdev_M;
reg [(DBITS-1):0] IHAdev_M, IRAdev_M, IDNdev_M;
reg TakeInt_dev_M;
always @(intr_devs or devnum or flushed_M or TakeInt_leg_M or ...) begin
  {OIEdev_M, IEdev_M }=
  {OIEinst_M,IEinst_M};
  {IHAdev_M, IRAdev_M, IDNdev_M }=
  {IHAinst_M,IRAinst_M,IDNinst_M};
  TakeInt_dev_M=TakeInt_inst_M;
  if(IEinst_M&&intr_devs&&!flushed_M) begin
    {OIEdev_M, IEdev_M, IRAdev_M, IDNdev_M}=
    {IEinst_M,1'b0, pcsave_M, devnum  };
    TakeInt_dev_M=1'b1;
  end
end
```
The next PC as computed by M-stage inst
- Similar to the pcgood
- But not quite the same!
  - pcgood is “next instruction should be fetched from here”
  - pcsave is “M-stage inst says we should fetch from here”
  - When are these not the same?
Why check for `flushed_M`?

- Prevents interrupt from being taken if instruction in M stage is a “bubble”
- What happens for ints if we don’t check?

```verilog
always @(intr_devs or devnum or flushed_M or TakeInt_leg_M or ...) begin
  ...
  if(IEinst_M&&intr_devs&&!flushed_M) begin
    {OIEdev_M,IEdev_M,IRAdev_M,IDNdev_M}=
    {IEinst_M,1'b0,pcsave_M,devnum};
    TakeInt_dev_M=1'b1;
  end
end
```
• Alternative approach
  – Write what happens to OM, what happens to IHA, etc.
    • Must consider all possible combinations of inst, iinst, and intreq
What’s Missing

• How to compute pcsave and pcgood?
  – Similar to before, just note the difference

• Which value does RSR read (e.g. for IE)?
  – Value from the beginning of the cycle (IE)?
  – After M-stage inst updates it (IEinst_M)?
  – After update from taking an interrupt (IEnew_M)?
Ha! Projects 4 and 5 are easy now!

• No, it’s just easier!
• Even one simple bug can take days to fix!
  – For interrupt-related bugs SignalTap will not help much
• Even with all this, plenty of ways to have a bug
  – Probably you’ll have several bugs
• Start working on Project 4 right now 😊
  – Even better, start working on it last week 😊