Review: Datapath for MIPS

- Use datapath figure to represent pipeline

1. Instruction Fetch
2. Decode/Register Read
3. Execute
4. Memory
5. Write Back

IFtch | Dcd | Exec | Mem | WB

IS $ | Reg | ALU | DS $ | Reg
Review: Problems for Computers

• Limits to pipelining: **Hazards** prevent next instruction from executing during its designated clock cycle
  - **Structural hazards**: HW cannot support this combination of instructions (single person to fold and put clothes away)
  - **Control hazards**: Pipelining of branches & other instructions **stall** the pipeline until the hazard; “**bubbles**” in the pipeline
  - **Data hazards**: Instruction depends on result of prior instruction still in the pipeline (missing sock)
Review: C.f. Branch Delay vs. Load Delay

• Load Delay occurs only if necessary (dependent instructions).

• Branch Delay always happens (part of the ISA).

• Why not have Branch Delay interlocked?
  • Answer: Interlocks only work if you can detect hazard ahead of time. By the time we detect a branch, we already need its value … hence no interlock is possible!
FYI: Historical Trivia

• First MIPS design did not interlock and stall on load-use data hazard

• Real reason for name behind MIPS: Microprocessor without Interlocked Pipeline Stages
  • Word Play on acronym for Millions of Instructions Per Second, also called MIPS
  • Load/Use ➔ Wrong Answer!
Outline

• Pipeline Control

• Forwarding Control

• Hazard Control
Piped Proc So Far …
New Representation: Regs more explicit

IF/DE.Ir = Instruction
DE/EX.A = BusA out of Reg
EX/ME.S = AluOut
EX/ME.D = Bus B pass-through for sw
ME/WB.S = ALuOut pass-through
ME/WB.M = Mem Result from lw
New Representation: Regs more explicit

What’s Missing???
Pipelined Processor (almost) for slides

Idea: Parallel Piped Control …
Pipelined Control

IR <- Mem[PC]; PC <- PC+4;

A <- R[rs]; B <- R[rt]

S <- A + B;
S <- A or ZX;
S <- A SX;
S <- A SX;

If Cond
PC <- PC+sx;

M <- Mem[S]
Mem[S] <- B

R[rd] <- S;
R[rt] <- S;
R[rd] <- M;

Equal

Reg. File

Mem Access

Exec

Data Mem

Reg. File
Data Stationary Control

- The Main Control generates the control signals during Reg/Dec
  - Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
  - Control signals for Mem (MemWr Branch) are used 2 cycles later
  - Control signals for Wr (MemtoReg MemWr) are used 3 cycles later
Let’s Try it Out

10 lw r1, 36(r2)
14 addl r2, r2, 3
20 sub r3, r4, r5
24 beq r6, r7, 100
28 ori r8, r9, 17
32 add r10, r11, r12

100 and r13, r14, 15
Start: Fetch 10

IF: 10 lw r1, 36(r2)
     14 addl r2, r2, 3
     20 sub r3, r4, r5
     24 beq r6, r7, 100
     30 ori r8, r9, 17
     34 add r10, r11, r12
     100 and r13, r14, 15
Fetch 14, Decode 10

Architectural pipeline stages:
- Fetch (IF)
- Decode (ID)
- Execute (EX)
- Memory (M)
- Write Back (WB)

Instruction window:
- Next PC
- PC

Instruction memory:
- IR

Register file:
- Reg

Memory:
- Mem

Control signals:
- MemCtrl
- WB Ctrl

Instruction window contents:
10 lw r1, 36(r2)
14 add r2, r2, 3
20 sub r3, r4, r5
24 beq r6, r7, 100
30 ori r8, r9, 17
34 add r10, r11, r12
100 and r13, r14, 15
Fetch 20, Decode 14, Exec 10

```
10 lw r1, 36(r2)
14 addl r2, r2, 3
20 sub r3, r4, r5
24 beq r6, r7, 100
30 ori r8, r9, 17
34 add r10, r11, r12
100 and r13, r14, 15
```
Fetch 30, Dcd 24, Ex 20, Mem 14, WB 10

Note Delayed Branch: always execute ori after beq
Fetch 100, Dcd 30, Ex 24, Mem 20, WB 14

```
10  lw  r1, 36(r2)
14  addl  r2, r2, 3
20  sub  r3, r4, r5
24  beq  r6, r7, 100
30  ori  r8, r9, 17
34  add  r10, r11, r12
100 and  r13, r14, 15
```
• Remember: \( \uparrow \) means triggered on edge.

• What is wrong here?
• Some signals are double clocked!

• In general: Inputs to edge components are their own pipeline regs

Watch out for stalls and such!
Administrivia

• Proj 2 – Due Sunday
• HW6 – Due Tuesday

• Midterm 2:
  • Friday, July 29: 11:00 – 2:00
  • Location TBD
  • If you are really so concerned about the drop deadline that this is a problem for you, talk to me about the possibility of taking the exam on Thursday
Megahertz Myth?
Outline

• Pipeline Control

• Forwarding Control

• Hazard Control
Fix by **Forwarding** result as soon as we have it to where we need it:

- **add** $t0, t1, t2$
- **sub** $t4, t0, t3$
- **and** $t5, t0, t6$
- **or** $t7, t0, t8$ *
- **xor** $t9, t0, t10$

* “or” hazard solved by register hardware
Forwarding

In general:

- For each stage i that has reg inputs
  - For each stage j after i that has reg output
    - If i.reg == j.reg → forward j value back to i.
    - Some exceptions ($0, invalid)

In particular:

- ALUInput ← (ALUResult, MemResult)
- MemInput ← (MemResult)
Pending Writes In Pipeline Registers

The diagram illustrates the pipeline stages of a computer processor. The stages include:

1. Instruction Address Unit (IAU)
2. NPC (Next Program Counter)
3. I Memory (I mem)
4. ALU (Arithmetic Logic Unit)
5. PC (Program Counter)
6. D Memory (D mem)
7. Registers (Regs)
8. Bus (B)
9. Address (A)
10. Immediate (im)
11. Op (Operation)
12. RW (Read/Write)
13. RS (Source Register)
14. RT (Target Register)
15. Instruction (op)

The diagram shows the flow of data and control signals through these pipeline stages, highlighting the processes involved in handling instructions and data.
Pending Writes In Pipeline Registers

- Current operand registers
- Pending writes
- hazard <=

\[
\begin{align*}
&((rs == \text{rw}_{\text{ex}}) \land \text{regW}_{\text{ex}}) \lor \\
&((rs == \text{rw}_{\text{mem}}) \land \text{regW}_{\text{me}}) \lor \\
&((rs == \text{rw}_{\text{wb}}) \land \text{regW}_{\text{wb}}) \lor \\
&((rt == \text{rw}_{\text{ex}}) \land \text{regW}_{\text{ex}}) \lor \\
&((rt == \text{rw}_{\text{mem}}) \land \text{regW}_{\text{mem}}) \lor \\
&((rt == \text{rw}_{\text{wb}}) \land \text{regW}_{\text{wb}})
\end{align*}
\]
Forwarding Muxes

- Detect nearest valid write op operand register and forward into op latches, bypassing remainder of the pipe

- Increase muxes to add paths from pipeline registers

- Data Forwarding = Data Bypassing
What about memory operations?

Tricky situation:

**MIPS:**
lw 0($t0)
sw 0($t1)

**RTL:**
R1 <- Mem[ R2 + 1 ];
Mem[R3+34] <- R1
What about memory operations?

Tricky situation:

MIPS:
\[ \text{lw } 0($t0) \]
\[ \text{sw } 0($t1) \]

RTL:
\[ \text{R1 } \leftarrow \text{Mem}[ \text{R2 + 1}] ; \]
\[ \text{Mem}[\text{R3+34}] \leftarrow \text{R1} \]

Solution:
Handle with bypass in memory stage!
Outline

• Pipeline Control

• Forwarding Control

• Hazard Control
Data Hazard: Loads (1/4)

- Forwarding works if value is available (but not written back) before it is needed. But consider …

\[
\text{lw } \$t0, 0(\$t1) \\
\text{sub } \$t3, \$t0, \$t2
\]

- Need result before it is calculated!
- Must stall use (sub) 1 cycle and then forward. …
Data Hazard: Loads (2/4)

- Hardware must stall pipeline
- Called "interlock"

```
lw $t0, 0($t1)
sub $t3,$t0,$t2
and $t5,$t0,$t4
or $t7,$t0,$t6
```
Data Hazard: Loads (3/4)

- Instruction slot after a load is called "load delay slot"

- If that instruction uses the result of the load, then the hardware interlock will stall it for one cycle.

- If the compiler puts an unrelated instruction in that slot, then no stall

- Letting the hardware stall the instruction in the delay slot is equivalent to putting a nop in the slot (except the latter uses more code space)
Data Hazard: Loads (4/4)

• Stall is equivalent to nop

lw $t0, 0($t1)

nop

sub $t3,$t0,$t2

and $t5,$t0,$t4

or $t7,$t0,$t6
Hazards / Stalling

In general:

- For each stage i that has reg inputs
  - If I’s reg is being written later on in the pipe but is not ready yet
    - Stages 0 to i: Stall (Turn CEs off so no change)
    - Stage i+1: Make a bubble (do nothing)
    - Stages i+2 onward: As usual

In particular:

- ALUoutput ← (MemResult)
Hazards / Stalling

Alternative Approach:

• Detect non-forwarding hazards in decode
  • Possible since our hazards are formal.
    - Not always the case.
  • Stalling then becomes:
    - Issue nop to EX stage
    - Turn off nextPC update (refetch same inst)
    - Turn off InstReg update (re-decode same inst)
Stall Logic

1. Detect non-resolving hazards.
   - 2a. Insert Bubble
   - 2b. Stall nextPC, IF/DE
Stall Logic

• Stall-on-issue is used quite a bit
  • More complex processors: many cases that stall on issue.
  • More complex processors: cases that can’t be detected at decode
    - E.g. value needed from mem is not in cache
      – proc must stall multiple cycles
By the way …

• Notice that our forwarding and stall logic is stateless!

• Big Idea: Keep it simple!
  • **Option 1**: Store old fetched inst in reg (“stall_temp”), keep state reg that says whether to use stall_temp or value coming off inst mem.
  • **Option 2**: Re-fetch old value by turning off PC update.