**Pipeline Implementation**

**IF stage**
- IR ← Mem[PC]
- NPC ← PC+4

**ID stage**
- A ← Regs[rs]
- B ← Regs[rt]
- Imm ← sign-extended immediate field of IR

**EX stage**
- ALUOutput ← A + Imm
- ALUOutput ← A func B
- ALUOutput ← A op Imm
- Branch
  - ALUOutput ← NPC + (Imm <<2)
  - Cond ← (A==0)

**MEM stage**
- For all instructions
  - PC ← NPC
- Memory reference
  - LMD ← Mem[ALUOutput]
  - Mem[ALUOutput] ← B
- Branch
  - If (cond) PC ← ALUOutput

**WB stage**
- If register-register/ALU/Load
  - Regs[rd] ← ALUOutput
- Register-immediate ALU
  - Regs[rt] ← LMD
Pipelined Implementation

IF stage
IF:ID.IR ← Mem[PC]
if((EX/MEM.opcode == branch) & EX/MEM.cond) then
  IF:ID.NPC ← EX/MEM.ALUOutput
else
  IF:ID.NPC ← PC+4

ID stage
ID/EX.A ← Regs[IF:ID.IR[rs]]; ID/EX.B ← Regs[IF:ID.IR[rt]]
ID/EX.NPC ← IF:ID.NPC; ID/EX.IR ← IF:ID.IR
ID/EX.Imm ← sign-extend(IF:ID.IR[Immediate])

EX stage
For all instructions
EX/MEM.IR ← ID/EX.IR
Memory reference
EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm
EX/MEM.B ← ID/EX.B
Register-register ALU/Register-immediate ALU
EX/MEM.ALUOutput ← ID/EX.A func ID/EX.B or
EX/MEM.ALUOutput ← ID/EX.A op ID/EX.Imm
Branch
EX/MEM.ALUOutput ← ID/EX.NPC + (ID/EX.Imm<<2)
EX/MEM.Cond ← (ID/EX.A==0)

MEM stage
For all instructions
MEM/WB.IR ← EX/MEM.IR
ALU instruction
MEM/WB.ALUOutput ← EX/MEM.ALUOutput
Memory reference
MEM/WB.LMD ← Mem[EX/MEM.ALUOutput] or
Mem[EX/MEM.ALUOutput] ← EX/MEM.B

WB stage
Register-register ALU/Register-immediate ALU
Regs[MEM/WB.IR[rd]] ← MEM/WB.ALUOutput or
Regs[MEM/WB.IR[rt]] ← MEM/WB.ALUOutput
Load
Regs[MEM/WB.IR[rt]] ← MEM/WB.LMD

Handling Hazards
There are no structural hazards in our pipeline
All data hazards can be detected (and some solved by forwarding) during ID phase
RAW(Read after Write) – instruction j reads the operand before instruction i writes it
WAW(Write after Write) – instruction j writes the operand before instruction i writes it
WAR(Write after Read) – instruction j writes the operand before instruction i reads it
RAR(Read after Read) is not a hazard
Load Hazards

No dependence
- Load R1, 45[R2]
- DADD R5, R6, R7
- DSUB R6, R6, R7
- OR R9, R6, R7

Dependence requiring stall
- Load R1, 45[R2]
- DADD R5, R1, R7
- DSUB R8, R6, R7
- OR R9, R6, R7

Dependence overcome by forwarding
- Load R1, 45[R2]
- DADD R5, R6, R7
- DSUB R8, R6, R7
- OR R9, R6, R7

Load Hazards that Require Stalls

Load
- Register-register ALU
  - IF/ID IR[rs] = = ID/EX IR[rt]
- Load, store, ALU imm, branch
  - IF/ID IR[rt] = = ID/EX IR[rt]

Data Hazards with Forwarding

Register-register ALU
- DADD R1, R2, R3
- DSUB R6, R6, R7
- OR R9, R6, R7
- Top ALU op-- EX/MEM ALU Output
- Bottom ALU op-- EX/MEM ALU Output

Register-immediate ALU
- DADD R5, R6, R7
- Top ALU op-- MEM/WB ALU Output
- Bottom ALU op-- MEM/WB ALU Output

Load
- MEM/WB IR[rt] = = ID/EX IR[rt]
- DADD R5, R6, R7
- Top ALU op-- MEM/WB ALU Output
- Bottom ALU op-- MEM/WB ALU Output

Control Hazards

If we only consider BEQZ and BNEZ (BEQ and BNE with R0) we can move comparison to the end of ID stage

To take advantage of that branch target needs to be computed early

We need additional adder

Only 1 clock cycle branch penalty

ALU followed by a branch on the result will incur stall
Control Hazards

IF stage

```
IF/ID.IR ← Mem[PC]
if((IF/ID.opcode == branch) & (Regs[(IF/ID.IR_6..10) op 0])) then
IF/ID.NPC ← IF/ID.NPC + (IF/ID.IR_16)
else
IF/ID.NPC ← PC+4
```

Dealing with Exceptions

Problem arises when instruction \(i+k\) raises an exception, while instruction \(i\) is being executed

Types of Exceptions:
- I/O request
- OS system call
- Tracing instruction execution
- Arithmetic overflow
- Page fault
- Memory protection violation
- etc.

Requirements

1. Synchronous vs. asynchronous
2. User requested vs. coerced
3. User maskable vs. nonmaskable
4. Within vs. between instructions
5. Resume vs. terminate

Difficult task is implementing exceptions within instructions that must resume after exception

Stopping and Restarting Execution

Saving the pipeline state:

- Force a trap instruction in the pipeline on next IF
- Until trap is taken turn off all writes for the faulting instruction and other instructions that follow in the pipeline
- When trap becomes active it saves PC of the faulting instruction, it will be used for return. If there are branches in pipeline, we should save PCs for \(branch\_delay+1\) instructions.

If the pipeline can be stopped so that instructions before faulting instruction are completed, and the others can be restarted it is said to have *precise exceptions*

Exception Handling in MIPS

Instruction \(j\) can cause exception before instruction \(i\) does

However we must handle exceptions the way we would have handled them without pipelining – first \(i\) and then \(j\)

Associate a status vector with the instruction

Turn off all writes if bit in status vector is set

When the instruction is in WB the status vector is checked and handled

Instruction Set Complications

Problem arises when an instruction can alter state early in the pipeline:

- Upon exception this state change must be undone
- Instructions that update memory are forced to work on registers, thus the state of partially completed instructions is in registers and can be saved and restored

If instruction set has very long instructions, pipelining is done at microinstruction level

\[\text{SUB}L2(R2, R3)\]
\[\text{MOV}C2 @(R1)(R2, 74(R2), R3)\]
Extending MIPS for FP Pipelining

FP operations are long and cannot be completed in 5 cycles (EX lasts more than 1 cycle)

Simply imagine that EX stage is duplicated for FP
There are multiple FP functional units

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction 1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>FP Instruction 1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
</tr>
<tr>
<td>Instruction 1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

It would be beneficial to pipeline EX stage for FP
We define:
Latency – number of cycles between the instruction that produces result and instruction that uses the result
Initiation interval – number of cycles that must elapse before issuing two operations of a given type

<table>
<thead>
<tr>
<th>Functional unit</th>
<th>Latency</th>
<th>Initiation interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer ALU</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Data memory</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>FP add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>FP/Int multiply</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>FP/Int divide</td>
<td>24</td>
<td>24</td>
</tr>
</tbody>
</table>

Hazards
Because DIV unit is not pipelined structural hazards can occur
Because instructions have varying running times number of register writes in a cycle can be >1
Instructions don’t reach WB in order, so WAW hazards are possible
Instructions can raise exceptions out of order
Stalls for RAW hazards will be longer due to long latency
WAW Hazards

- Although it seems useless sequence of instructions (STORE overwrites F2 immediately after ADD writes it) we must detect WAW hazard and make sure the later value appears in register.
- One approach (shown) is to delay issue of later instruction.
- Another approach is to stamp the result of ADD and don’t write it into memory.

Write Port Structural Hazards

- Detect write port hazard in ID stage, use shift register that indicates when already issued instructions will use write port, shift reservation register one bit at each clock cycle.
- Alternative stall before MEM or WB.

Detecting and Handling Hazards

- Since FP and integer operations use different registers we need only consider moves and loads/stores.
- Pipeline checks in ID for:
  - Structural hazards – DIV unit and write port.
  - RAW hazards – wait until source registers are not listed as pending destinations.
  - WAW hazards – determine if any instruction in EX stage has the same destination register, if so stall the current instruction.

Maintaining Precise Exception

- Out-of-order completion:
  - Buffer results of the operation until all previous operations have completed.
  - History file keeps track of original values of registers.
  - Future file keeps track of new values, registers are updated when all previous instructions have completed.
  - Proceed only if sure that no previous instructions will cause exceptions.

Homework

- Due Tuesday, October 12 by the end of the class.
- Submit either in class (paper) or by E-mail (PS or PDF only) or bring the paper copy to my office.
- Do exercises A.1 (assume branch target and condition are calculated in ID stage), A.5, A.6 and A.7.