CS161 – Design and Architecture of Computer Systems

Pipelined CPU Design
Pipelining Analogy

- Laundry steps:
  - Wash
  - Dry
  - Fold
  - Put it away – Closet / Dresser / Neat pile on floor
Pipelining Analogy

Assuming each step take 1 hour, 4 loads would take 16 hours!
Pipelining Analogy

To speed things up, overlap steps

1

2

3

4

4 loads of laundry now only takes 7 hours!
Speedup of Pipelining

- $k$ stages pipeline, $t$ time per stage, $n$ jobs
- Non-pipelined time = $n \times k \times t$
- Pipelined time = $(k+n-1) \times t$

$$Speedup = \frac{n \times k \times t}{(k+n-1) \times t} \xrightarrow[n \to \infty]{k}$$

This is an ideal case:
- No job depends on a previous job
- All jobs behave exactly the same

Is life always this beautiful?
Pipelining Multiple Loads of Laundry: In Practice

the slowest step decides throughput

Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
Single Cycle, Multiple Cycle, vs. Pipeline

Single Cycle Implementation:

Load

Store

Waste

Multiple Cycle Implementation:

Load

Store

R-type

Pipeline Implementation:

Load

Store

R-type
MIPS Pipeline

- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register
Pipeline Performance

**Single-cycle ($T_c = 800\text{ps}$)**

- Time: 200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800
- Program execution order (in instructions):
  - `lw $1, 100(0)`
  - `lw $2, 200(0)`
  - `lw $3, 300(0)`

**Pipelined ($T_c = 200\text{ps}$)**

- Time: 200, 400, 600, 800, 1000, 1200, 1400
- Program execution order (in instructions):
  - `lw $1, 100(0)`
  - `lw $2, 200(0)`
  - `lw $3, 300(0)`
Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time

  \[
  \text{Time between instructions - pipelined} = \frac{\text{Time between instructions - non-pipelined}}{\text{Number of pipeline stages}}
  \]

- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease
Pipelining and ISA Design

- MIPS ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode
    - c.f. x86: 1- to 17-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in 3rd stage, access memory in 4th stage
  - Alignment of memory operands
    - Memory access takes only one cycle
Pipeline Summary

The BIG Picture

- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Instruction set design affects complexity of pipeline implementation
MIPS Pipelined Datapath

Right-to-left flow leads to hazards
Pipeline registers

- Need registers between stages
  - To hold information produced in previous cycle
Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
  - “Single-clock-cycle” pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - c.f. “multi-clock-cycle” diagram
    - Graph of operation over time
- We’ll look at “single-clock-cycle” diagrams for load & store
Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle

<table>
<thead>
<tr>
<th></th>
<th>State</th>
<th>State</th>
<th>State</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $14, $5, $6</td>
<td>IF/ID</td>
<td>ID/EX</td>
<td>EX/MEM</td>
<td>MEM/WB</td>
</tr>
<tr>
<td>lw $13, 24 ($1)</td>
<td>PC</td>
<td>Address</td>
<td>Instruction memory</td>
<td>Instruction memory</td>
</tr>
<tr>
<td>add $12, $3, $4</td>
<td>Read register 1</td>
<td>Read register 2</td>
<td>Write data</td>
<td>Write data</td>
</tr>
<tr>
<td>sub $11, $2, $3</td>
<td>Read data 1</td>
<td>Read data 2</td>
<td>ALU</td>
<td>Zero ALU result</td>
</tr>
<tr>
<td>lw $10, 20($1)</td>
<td>Sign extend</td>
<td>16</td>
<td>32</td>
<td>PC</td>
</tr>
</tbody>
</table>

- Instruction fetch
- Instruction decode
- Execution
- Memory
- Write-back
IF for Load, Store, ...
ID for Load, Store, ...
EX for Load
MEM for Load
WB for Load

Wrong register number
Corrected Datapath for Load
EX for Store
MEM for Store
All instruction classes must follow the same path and timing through the pipeline stages.

Any performance impact?
Multi-Cycle Pipeline Diagram

- Form showing resource usage

Why Pipeline?
Because the resources are there! 😊

Program execution order (in instructions):

- `lw $10, 20($1)`
- `sub $11, $2, $3`
- `add $12, $3, $4`
- `lw $13, 24($1)`
- `add $14, $5, $6`

Time (in clock cycles):
CC 1  CC 2  CC 3  CC 4  CC 5  CC 6  CC 7  CC 8  CC 9

steady state (full pipeline)
Multi-Cycle Pipeline Diagram

Traditional form
Pipelined Control

- Control signals derived from instruction, in ID stage
  - As in single-cycle implementation
Pipelined Control
Hazards

- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy: why we have I$ and D$
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction
Structure Hazards

- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches
Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - add $s0, $t0, $t1
  - sub $t2, $s0, $t3
General framework of data hazards

- **Range and Domain of Instructions**
  - \( D(i) \): domain of \( i \) = all values read by \( i \) prior to execution
  - \( R(i) \): range of instruction \( i \) = all values written by \( i \), after execution

- **Read After Write (RAW), true, or dataflow, dependence**
  - \( i_1 \): add \( r_1, r_2, r_3 \)
  - \( i_2 \): add \( r_4, r_1, r_5 \)

- **Write After Read (WAR), anti dependence**
  - \( i_1 \): add \( r_1, r_2, r_3 \)
  - \( i_2 \): add \( r_2, r_4, r_5 \)

- **Write After Write (WAW), output dependence**
  - \( i_1 \): add \( r_1, r_2, r_3 \)
  - \( i_2 \): add \( r_1, r_4, r_5 \)

\[
\begin{align*}
RAW : R(i_1) \cap D(i_2) &\neq \emptyset \\
WAR : D(i_1) \cap R(i_2) &\neq \emptyset \\
WAW : R(i_1) \cap R(i_2) &\neq \emptyset
\end{align*}
\]
WAR & WAW

- WAR & WAW are name dependencies
  - Dependence is on the container’s name not on the value contained.
  - Can be eliminated by renaming, static (in software) or dynamic (in hardware)

- WAW & WAR cannot occur in the 5-stage pipeline
  - All the writing happens in WB stage, in order issue of all instructions
Forwarding (aka Bypassing)

- Use result when it is computed
- Don’t wait for it to be stored in a register
- Requires extra connections in the datapath
Load-Use Data Hazard

▷ Can’t always avoid stalls by forwarding
  ▷ If value not computed when needed
  ▷ Can’t forward backward in time!
Reorder code to avoid use of load result in the next instruction

- C code for \( A = B + E; \ C = B + F; \)
- Reordering is commonly done by modern compilers

```assembly
lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
lw $t4, 8($t0)
```

- 13 cycles

```assembly
lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
lw $t4, 8($t0)
add $t5, $t1, $t4
sw $t5, 16($t0)
```

- 11 cycles
Data Hazards in ALU Instructions

- Consider this sequence:
  
  ```
  sub $2, $1,$3
  and $12,$2,$5
  or  $13,$6,$2
  add $14,$2,$2
  sw  $15,100($2)
  ```

- We can resolve hazards with forwarding
  
  - How do we detect when to forward?
Dependencies & Forwarding

Program execution order (in instructions)

- `sub $2, $1, $3`
- `and $12, $2, $5`
- `or $13, $6, $2`
- `add $14, $2, $2`
- `sw $15, 100($2)`
Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register

- ALU operand register numbers in EX stage are given by
  - ID/EX.RegisterRs, ID/EX.RegisterRt

- Data hazards when
  1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
  If dest == either src register
  Fwd from EX/MEM pipeline reg
  1b. EX/MEM.RegisterRd = ID/EX.RegisterRt

  2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
  If dest == src register
  Fwd from MEM/WB pipeline reg
  2b. MEM/WB.RegisterRd = ID/EX.RegisterRt
Detecting the Need to Forward

- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not $zero
  - EX/MEM.RegisterRd ≠ 0,
    MEM/WB.RegisterRd ≠ 0
Forwarding Paths

b. With forwarding
Forwarding Conditions

- **EX hazard**
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRd))
    ForwardA = 10
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10

- **MEM hazard**
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01
Double Data Hazard

- Consider the sequence:
  - add $1, $1, $2
  - add $1, $1, $3
  - add $1, $1, $4

- Both hazards occur
  - Want to use the most recent

- Revise MEM hazard condition
  - Only fwd if EX hazard condition isn’t true
Revised Forwarding Condition

MEM hazard

- if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
  and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
  and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
  ForwardA = 01

- if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
  and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
  and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
  ForwardB = 01
Datapath with Forwarding
Load-Use Data Hazard

Need to stall for one cycle
Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs, IF/ID.RegisterRt
- Load-use hazard when
  - ID/EX.MemRead and
    - ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
    - (ID/EX.RegisterRt = IF/ID.RegisterRt))
- If detected, stall and insert bubble
How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX, MEM and WB do *nop* (no-operation)

- Prevent update of PC and IF/ID register
  - Current instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for `lw`
    - Can subsequently forward to EX stage
Stall/Bubble in the Pipeline

Program execution order (in instructions)

- `lw $2, 20($1)`
- and becomes `nop`
- and `$4, $2, $5`
- or `$8, $2, $6`
- `add $9, $4, $2`

Stall inserted here
Stall/Bubble in the Pipeline

Program execution order (in instructions)

Iw $2, 20($1)

and becomes nop

and $4, $2, $5 stalls in ID

or $8, $2, $6 stalled in IF

add $9, $4, $2

Or, more accurately...
Datapath with Hazard Detection
Stalls and Performance

The BIG Picture

- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure
Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can’t always fetch correct instruction
    - Still working on ID stage of branch

- In MIPS pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage
Pipelined Control (Simplified)
Branch Hazards

- If branch outcome resolved in MEM

Time (in clock cycles)

<table>
<thead>
<tr>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
<th>CC 7</th>
<th>CC 8</th>
<th>CC 9</th>
</tr>
</thead>
</table>

Program execution order (in instructions)

40 beq $1, $3, 28

44 and $12, $2, $5

48 or $13, $6, $2

52 add $14, $2, $2

72 lw $4, 50($7)

Flush these instructions (Set control values to 0)

PC
Reducing Branch Delay

- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator

- Example: branch taken

```
36:  sub  $10, $4, $8
40:  beq  $1,  $3,  7
44:  and  $12, $2, $5
48:  or   $13, $2, $6
52:  add  $14, $4, $2
56:  slt  $15, $6, $7
    ...
72:  lw    $4, 50($7)
```
Example: Branch Taken

and $12, $2, $5

beq $1, $3, 7

sub $10, $4, $8

before<1>

before<2>
Example: Branch Taken

IF, Flush

lw $4, 50($7)

Bubble (nop)

beq $1, $3, 7

sub $10, ...

before<1>

Clock 4
Data Hazards for Branches

- If a comparison register is a destination of 2\textsuperscript{nd} or 3\textsuperscript{rd} preceding ALU instruction

- Resolve using forwarding
Data Hazards for Branches

- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
  - Need 1 stall cycle

```
lw   $1, addr
add  $4, $5, $6
beq  stalled
beq  $1, $4, target
```
Data Hazards for Branches

- If a comparison register is a destination of immediately preceding load instruction
  - Need 2 stall cycles

```plaintext
lw  $1, addr
beq stalled
beq stalled
beq $1, $0, target
```
1. Stall on Branch
   - Simplest way to handle control hazard: stall
   - Wait until branch outcome determined before fetching next instruction

2. Delay Slots Instructions
   - Let the CPU do useful work while waiting for the branch target and condition
     - Execute instructions that are independent of the branch

3. Predict Branch
   - Instead of stalling and waiting for branch outcome, predict branch and execute
     - If incorrect, flush pipeline and take correct path
Branch Delay Slots

- Use N *branch delay slot* to eliminate N stalls after branches.

- The N instruction after a conditional branch is *always executed*, regardless of whether the branch is taken or not!

- Reduced delay branch of MIPS (determine outcome of branch on ID stage) have 1 branch delay slot.
Filling the branch delay slot

- The branch delay slot is only useful if you can find something to put there.
- If you can’t find anything, you must put a *nop* to insure correctness.

```
add $R1,$R2,R3
sub $R5,$R7,R8
beq $R1,$R2,L
add $R4,$R4,R4
...
L:   add $R2, $R10, $R11
```
Branch Prediction

- Longer pipelines can’t readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay
MIPS with Predict Not Taken

Assumes branch in ID

Prediction correct

Program execution order (in instructions)

Time

200 400 600 800 1000 1200 1400

add $4, $5, $6
beq $1, $2, 40
lw $3, 300($0)

200 ps

Instruction fetch
Reg
ALU
Data access
Reg

Prediction incorrect

Program execution order (in instructions)

Time

200 400 600 800 1000 1200 1400

add $4, $5, $6
beq $1, $2, 40
lw $3, 300($0)

200 ps

Instruction fetch
Reg
ALU
Data access
Reg

or $7, $8, $9

400 ps

Instruction fetch
Reg
ALU
Data access
Reg
More-Realistic Branch Prediction

- **Static branch prediction**
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken

- **Dynamic branch prediction**
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history
Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
  - Branch prediction buffer (aka branch history table)
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken)
  - To execute a branch
    - Check table, expect the same outcome
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction
Dynamic Branch Prediction

- Branch Prediction Buffer (BPB) accessed with Instruction on I-Fetch

- Also called Branch History Table (BHT), Branch Prediction Table (BPT)
1-bit Predictor

- Each BHT entry is 1-bit
  - Bit records last outcome of the branch
  - Predicts that next outcome is the same as the last

- Loop1: ---
  ---
  Loop2: ---
  ---
  BEZ R2, Loop2
  ---
  BNEZ R3, Loop1

- BEZ always mispredicted twice for every loop
  - Once on entry and once on exit
2-bit Predictor

- Prediction must miss twice before it is changed

- 2-bit BHT
- Also called 2-bit saturating counter
- Can be extended to N-bits (typically N=2)
2-bit predictor

- Loop1: ---
  ---
- Loop2: ---
  ---
  BEZ R2, Loop2
  ---
  BNEZ R3, Loop1
Calculating the Branch Target

- Even with predictor, still need to calculate the target address
  - 1-cycle penalty for a taken branch

- Branch target buffer (BTB)
  - Cache of target addresses
  - Indexed by PC when instruction fetched
    - If hit and instruction is branch predicted taken, can fetch target immediately
Branch Target Buffers (BTB)

- Branch target calculation is costly and stalls instruction fetch
- BTB enable fetching to begin after IF-stage
- BTB cache predicted PC value
Exceptions and Interrupts

“Unexpected” events requiring change in flow of control

- Different ISAs use the terms differently

Exception
- Arises within the CPU
  - e.g., undefined opcode, overflow, syscall, …

Interrupt
- From an external I/O controller

Dealing with them without sacrificing performance is hard
Handling Exceptions

- In MIPS, exceptions managed by a System Control Coprocessor (CP0)
- Save PC of offending (or interrupted) instruction
  - In MIPS: Exception Program Counter (EPC)
- Save indication of the problem
  - In MIPS: Cause register
  - We’ll assume 1-bit
    - 0 for undefined opcode, 1 for overflow
- Jump to handler at 8000 00180
Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
  - Take corrective action
  - use EPC to return to program
- Otherwise
  - Terminate program
  - Report error using EPC, cause, …
An Alternate Mechanism

- Vectored Interrupts
  - Handler address determined by the cause

Example:
- Undefined opcode: \( \text{C000 0000} \)
- Overflow: \( \text{C000 0020} \)
- ...: \( \text{C000 0040} \)
Exceptions in a Pipeline

- Another form of control hazard
- Consider overflow on add in EX stage
  
  \[ \text{add } \$1, \$2, \$1 \]

  - Complete previous instructions
  - Flush add and subsequent instructions
  - Set Cause and EPC register values
  - Transfer control to handler

- Similar to mispredicted branch
  - Use much of the same hardware
Pipeline with Exceptions
Exception Properties

- Restartable exceptions
  - Pipeline can flush the instruction
  - Handler executes, then returns to the instruction
    - Refetched and executed from scratch

- PC saved in EPC register
  - Identifies causing instruction
  - Actually PC + 4 is saved
    - Handler must adjust
Exception Example

Exception on `add` in

40 sub $11, $2, $4
44 and $12, $2, $5
48 or $13, $2, $6
4c add $1, $2, $1
50 slt $15, $6, $7
54 lw $16, 50($7)
...

Handler

80000180 sw $25, 1000($0)
80000184 sw $26, 1004($0)
...

Exception Example

lw $16, 50($7)
slt $15, $6, $7
add $1, $2, $1
or $13, ...
and $12, ...

Clock 6
Exception Example

sw $25, 1000($0)

bubble (nop)

bubble

bubble

or $13, ...
Multiple Exceptions

- Pipelining overlaps multiple instructions
  - Could have multiple exceptions at once
- Simple approach: deal with exception from earliest instruction
  - Flush subsequent instructions
  - “Precise” exceptions
- In complex pipelines
  - Multiple instructions issued per cycle
  - Out-of-order completion
  - Maintaining precise exceptions is difficult!
Imprecise Exceptions

- Just stop pipeline and save state
  - Including exception cause(s)
- Let the handler work out
  - Which instruction(s) had exceptions
  - Which to complete or flush
- Simplifies hardware, but more complex handler software
- Not feasible for complex multiple-issue out-of-order pipelines
Fallacies

- Pipelining is easy (!)
  - The basic idea is easy
  - The devil is in the details
    - e.g., detecting data hazards

- Pipelining is independent of technology
  - So why haven’t we always done pipelining?
  - More transistors make more advanced techniques feasible
Pitfalls

- Poor ISA design can make pipelining harder
  - e.g., complex instruction sets (VAX, IA-32)
    - Significant overhead to make pipelining work
    - IA-32 micro-op approach
  - e.g., complex addressing modes
    - Register update side effects, memory indirection
Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control