Review: Single Cycle vs. Multiple Cycle Timing

Single Cycle Implementation:

Multiple Cycle Implementation:

How Can We Make It Even Faster?

- Split the multiple instruction cycle design into smaller and smaller steps
  - There is a point of diminishing returns where as much time is spent loading the state registers as doing the work
- Start fetching and executing the next instruction before the current one has completed
  - Pipelining – (all?) modern processors are pipelined for performance
  - Superpipelining – many pipeline stages, very fast clock
- Fetch (and execute) more than one instruction at a time (out-of-order superscalar and VLIW (epic))
- Fetch (and execute) instructions from more than one instruction stream (multithreading (hyperthreading))
A Pipelined MIPS Processor

- Start the next instruction before the current one has completed
  - Improves throughput - total amount of work done in a given time
  - Instruction latency (execution time, delay time, response time - time from the start of an instruction to its completion) is not reduced

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle 6</th>
<th>Cycle 7</th>
<th>Cycle 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>IFetch</td>
<td>Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td>IFetch</td>
<td>Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-type</td>
<td>IFetch</td>
<td>Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Clock cycle (pipeline stage time) is limited by the slowest stage
- For some instructions, some stages are wasted cycles

Single Cycle, Multiple Cycle, vs. Pipeline

Single Cycle Implementation:

- Cycle 1
- Cycle 2
- lw
- sw

Multiple Cycle Implementation:

- Cycle 1
- Cycle 2
- Cycle 3
- Cycle 4
- Cycle 5
- Cycle 6
- Cycle 7
- Cycle 8
- Cycle 9
- Cycle 10

Pipeline Implementation:

- lw
- sw
- R-type

Pipeline clock same as multi-cycle clock
MIPS Pipeline Datapath Modifications

- What do we need to add/modify in our MIPS datapath?
  - State registers between each pipeline stage to isolate them

MIPS Pipeline Control Path Modifications

- All control signals can be determined during Decode and held in the state registers between pipeline stages
Pipelining the MIPS ISA

- What makes it easy
  - all instructions are the same length (32 bits)
    - can fetch in the 1st stage and decode in the 2nd stage
  - few instruction formats (three) with symmetry across formats
    - can begin reading register file in 2nd stage
  - memory operations can occur only in loads and stores
    - can use the execute stage to calculate memory addresses
  - each MIPS instruction writes at most one result (i.e., changes the machine state) and does so near the end of the pipeline (MEM and WB)

- What makes it hard
  - structural hazards: what if we had only one memory?
  - control hazards: what about branches?
  - data hazards: what if an instruction's input operands depend on the output of a previous instruction?

Graphically Representing MIPS Pipeline

- Can help with answering questions like:
  - How many cycles does it take to execute this code?
  - What is the ALU doing during cycle 4?
  - Is there a hazard, why does it occur, and how can it be fixed?
Why Pipeline? For Performance!

Once the pipeline is full, one instruction is completed every cycle so CPI = 1

Can Pipelining Get Us Into Trouble?

- Yes: Pipeline Hazards
  - structural hazards: attempt to use the same resource by two different instructions at the same time
  - data hazards: attempt to use data before it is ready
    - An instruction’s source operand(s) are produced by a prior instruction still in the pipeline
  - control hazards: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
    - branch and jump instructions, exceptions

- Can always resolve hazards by waiting
  - pipeline control must detect the hazard
  - and take action to resolve hazards
A Single Memory Would Be a Structural Hazard

Time (clock cycles)

- lw
- Inst 1
- Inst 2
- Inst 3
- Inst 4

Reading data from memory

- Can fix with separate instr and data memories

How About Register File Access?

Time (clock cycles)

- add $1, IM, Reg
- Inst 1
- Inst 2
- add $2, $1, IM, Reg

Reading instruction from memory
How About Register File Access?

Fix register file access hazard by doing reads in the second half of the cycle and writes in the first half.

clock edge that controls register writing

clock edge that controls loading of pipeline state registers

Register Usage Can Cause Data Hazards

- Dependencies backward in time cause hazards

- Read before write data hazard
Register Usage Can Cause Data Hazards

- Dependencies backward in time cause hazards

- Read before write data hazard

Loads Can Cause Data Hazards

- Dependencies backward in time cause hazards
Loads Can Cause Data Hazards

- Dependencies backward in time cause hazards

```
lw $1,4($2)           IM  |  IM  |  DM  |  Reg
sub $4,$1,$5          IM  |  IM  |  DM  |  Reg
and $6,$1,$7          IM  |  IM  |  DM  |  Reg
or $8,$1,$9           IM  |  IM  |  DM  |  Reg
xor $4,$1,$5          IM  |  IM  |  DM  |  Reg
```

- Load-use data hazard

One Way to “Fix” a Data Hazard

```
add $1,                IM  |  IM  |  DM  |  Reg
stall                 IM  |  IM  |  DM  |  Reg
stall                 IM  |  IM  |  DM  |  Reg
sub $4,$1,$5          IM  |  IM  |  DM  |  Reg
and $6,$1,$7          IM  |  IM  |  DM  |  Reg
```

Can fix data hazard by waiting - stall
Another Way to “Fix” a Data Hazard

Fix data hazards by forwarding results as soon as they are available to where they are needed.

Instr. Order

add $1, sub $4,$1,$5
and $6,$1,$7
or $8,$1,$9
xor $4,$1,$5

Fix data hazards by forwarding results as soon as they are available to where they are needed.
Forwarding with Load-use Data Hazards

Instr. Order

lw $1,4($2)  
sub $4,$1,$5  
and $6,$1,$7  
or $8,$1,$9  
xor $4,$1,$5

Will still need one stall cycle even with forwarding
Control Hazards

- When the flow of instruction addresses is not sequential (i.e., \( PC = PC + 4 \))
  - Conditional branches (\( \text{beq}, \text{bne} \))
  - Unconditional branches (\( \text{j}, \text{jal}, \text{jr} \))
  - Exceptions

- Possible “solutions”
  - Stall (impacts performance)
  - Move branch decision point as early in the pipeline as possible, thereby reducing the number of stall cycles
  - Delay decision (requires compiler support)
  - Predict and hope for the best!

- Control hazards occur less frequently than data hazards, but there is *nothing* as effective against control hazards as forwarding is for data hazards

Jumps Incur One Stall

- Jumps not decoded until ID, so one flush is needed
  - To flush, set IF.Flush to zero the instruction field of the IF/ID pipeline register (turning it into a *noop*)

- Fortunately, jumps are very infrequent – only 3% of the SPECint instruction mix
Review: MIPS Pipeline Control & Datapath

- All control signals can be determined during Decode and held in the state registers between pipeline stages.

Branches Cause Control Hazards

- Dependencies backward in time cause hazards.
One Way to “Fix” a Branch Control Hazard

- Instr. Order

  - beq
  - flush
  - flush
  - beq target
  - Inst 3

Fix branch hazard by waiting – but affects CPI

Another Way to “Fix” a Branch Control Hazard

- Move branch decision hardware back to as early in the pipeline as possible – i.e., during the decode cycle

- Instr. Order

  - beq
  - flush
  - beq target
  - Inst 3

Fix branch hazard by waiting – flush
Yet Another Way to “Fix” a Control Hazard

- “Predict branches are always not taken – and take corrective action when wrong (i.e., taken)

4 beq $1,$2,2
8 sub $4,$1,$5
16 and $6,$1,$7
20 or r8,$1,$9

- To flush, set IF.Flush to zero the instruction field of the IF/ID pipeline register (turning it into a \textit{noop})

Two “Types” of Stalls

- \textit{Noop} instruction (or bubble) inserted between two instructions in the pipeline (e.g., load-use hazards)
  - Keep the instructions \textit{earlier} in the pipeline (later in the code) from progressing down the pipeline for a cycle (“bounce” them in place with write control signals)
  - Insert \textit{noop} instruction by zeroing control bits in the pipeline register at the appropriate stage
  - Let the instructions later in the pipeline (earlier in the code) progress normally down the pipeline

- Flushes (or instruction squashing) where an instruction in the pipeline is \textit{replaced} with a \textit{noop} instruction (as done for instructions located sequentially after \texttt{j} and \texttt{beq} instructions)
  - Zero the control bits for the instruction to be flushed
Many Other Pipeline Structures Are Possible

- What about the (slow) multiply operation?
  - Make the clock twice as slow or …
  - Let it take two cycles (since it doesn’t use the DM stage)

- What if the data memory access is twice as slow as the instruction memory?
  - Make the clock twice as slow or …
  - Let data memory access take two cycles (and keep the same clock rate)

Pipelining Summary

- All modern day processors use pipelining
- Pipelining doesn’t help latency of single task; it helps throughput of entire workload
- Potential speedup: a really fast clock cycle and able to complete one instruction every clock cycle (CPI)
- Pipeline rate limited by slowest pipeline stage
  - Unbalanced pipe stages makes for inefficiencies
  - The time to “fill” pipeline and time to “drain” it can impact speedup for deep pipelines and short code runs
- Must detect and resolve hazards
  - Stalling negatively affects CPI (makes CPI greater than the ideal of 1)