Processor Design – Pipelined Processor (II)

Hung-Wei Tseng
Announcement

- Homework #2 due next Tuesday!
- No office hour of Hung-Wei this Friday
  - Office hour next Monday
  - Check the website and calendar
Recap: Pipelining

- Break up the logic with “pipeline registers” into pipeline stages
  - Each pipeline registers is clocked
  - Each pipeline stage takes one cycle
  - Pipeline registers only updates at the end of a clock
- Each stage acts on a different instruction simultaneously
- States/control signals of instructions are hold in pipeline registers
5-stage pipelined processor
Simplified pipeline diagram

- Use symbols to represent the physical resources with the abbreviations for pipeline stages.
  - IF, ID, EXE, MEM, WB
- Horizontal axis represent the timeline, vertical axis for the instruction stream
- Example:
  - `add $1, $2, $3`
  - `lw $4, 0($5)`
  - `sub $6, $7, $8`
  - `sub $9, $10, $11`
  - `sw $1, 0($12)`
Even though we perfectly divide pipeline stages, it’s still hard to achieve $\text{CPI} == 1$.

Pipeline hazards:
- Structural hazard
  - The hardware does not allow two pipeline stages to work concurrently
- Data hazard
  - A later instruction in a pipeline stage depends on the outcome of an earlier instruction in the pipeline
- Control hazard
  - The processor is not clear about what’s the next instruction to fetch
Structural hazard

• The hardware cannot support the combination of instructions that we want to execute at the same cycle.
• The original pipeline incurs structural hazard when two instructions competing the same register.
• Solution: write early, read late
  • Writes occur at the clock edge and complete long enough before the end of the clock cycle.
  • This leaves enough time for outputs to settle for reads.
  • The revised register file is the default one from now!

```
add $1, $2, $3
lw $4, 0($5)
sub $6, $7, $8
sub $9,$10, $1
sw $1, 0($12)
```
What pair of instructions will be problematic if we allow R-type instructions to skip the “MEM” stage?

A: a & b
B: a & c
C: b & e
D: c & e
E: None
Structural hazard

- The design of hardware causes structural hazard
- We need to modify the hardware design to avoid structural hazard
Data hazard
What just happened here is problematic for the following instructions in our current pipeline?

```
add $1, $2, $3
lw $4, 0($1)
sub $6, $7, $8
sub $9,$10, $1
sw $1, 0($12)
```

A. The register file and memory are both active at the same cycle
B. The ALU and data memory are both active at the same cycle
C. A value is used before it’s produced
D. Both A and B
E. Both A and C
Data hazard

- When an instruction in the pipeline needs a value that is not available
- Data dependences
  - The output of an instruction is the input of a later instruction
  - May result in data hazard if the later instruction that consumes the result is still in the pipeline
Data dependences

- How many pairs of data dependences are there in the following code?

```
add $1, $2, $3
lw  $4, 0($1)
sub $5, $2, $4
sub $1, $3, $1
sw  $1, 0($5)
```

No every “data dependency” will lead to “data hazards”.

A. 1
B. 2
C. 3
D. 4
E. 5
Sol. of data hazard I: Stall

- When the source operand of an instruction is not ready, stall the pipeline
  - Suspend the instruction and the following instruction
  - Allow the previous instructions to proceed
  - This introduces a pipeline bubble: a bubble does nothing, propagate through the pipeline like a nop instruction

- How to stall the pipeline?
  - Disable the PC update
  - Disable the pipeline registers on the earlier pipeline stages
  - When the stall is over, re-enable the pipeline registers, PC updates
Hazard detection & stall

Check if the destination register of EX == source register of the instruction in ID

Check if the destination register of MEM == source register of the instruction in ID

Insert a “noop” if we need to stall
Performance of stall

15 cycles! CPI == 3
(If there is no stall, CPI should be just 1!)

```
add $1, $2, $3
lw  $4, 0($1)
sub $5, $2, $4
sub $1, $3, $1
sw  $1, 0($5)
```
Sol. of data hazard II: Forwarding

• The result is available after EXE and MEM stage, but publicized in WB!
• The data is already there, we should use it right away!
• Also called bypassing

```
add $1, $2, $3
lw  $4, 0($1)
sub $5, $2, $4
sub $1, $3, $1
sw  $1, 0($5)
```

We can obtain the result here!
Sol. of data hazard II: Forwarding

- Take the values, where ever they are!

```assembly
add $1, $2, $3
lw  $4, 0($1)
sub $5, $2, $4
sub $1, $3, $1
sw  $1, 0($5)
```

10 cycles! CPI == 2 (Not optimal, but much better!)
Design a forwarding unit

• How many of the following inputs are required for forwarding the result from the previous instruction (Ins#1) to the EXE stage of the current instruction (Ins#2)?

  • Rd of Ins#2
  • Rs of Ins#2
  • Rt of Ins#2
  • ReadData 2 of Ins #2
  • Rd of Ins#1
  • Rs of Ins#1
  • Rt of Ins#1
  • ReadData 2 of Ins #1
  • Control signals of Ins #1

A. 5  
B. 6  
C. 7  
D. 8  
E. 9  

We need to know the following:

1. If the ins#1 update a register (RegWrite)
2. If the destination register of ins #1 (rt, td) is a source of ins #2
   If ins #1 is R-type: rs, rt of ins #2 == rd of ins #1
   If ins #1 is I-type: rs, rt of ins #2 == rt of ins #1
When can/should we forward data?

- If the instruction entering the EXE stage consumes a result from a previous instruction that is entering MEM stage or WB stage
  - A source of the instruction entering EXE stage is the destination of an instruction entering MEM/WB stage
  - The previous instruction must be an instruction that updates register file
Forwarding in hardware

How about load?
There is still a case that we have to stall...

- Revisit the following code:

  ```
  add  $1, $2, $3
  lw   $4, 0($1)
  sub  $5, $2, $4
  sub  $1, $3, $1
  sw   $1, 0($5)
  ```

  If an instruction in EX/MEM updates a register (RegWrite)
  If an instruction in EX/MEM reads memory (MemRead)
  If the destination register of EX/MEM is a source of ID/EX (rs, rt of ID/EX == rt of EX/MEM #1)

- If the instruction entering EXE stage depends on a load instruction that does not finish its MEM stage yet, we have to stall!

- We call this hazard detection

We need to know the following:
1. If an instruction in EX/MEM updates a register (RegWrite)
2. If an instruction in EX/MEM reads memory (MemRead)
3. If the destination register of EX/MEM is a source of ID/EX (rs, rt of ID/EX == rt of EX/MEM #1)
Hazard detection with forwarding
Control hazard
Control hazard

• Consider the following code and the pipeline we designed:

```assembly
LOOP: lw $t3, 0($s0)
      addi $t0, $t0, 1
      add $v0, $v0, $t3
      addi $s0, $s0, 4
      bne $t1, $t0, LOOP
      sw $v0, 0($s1)
```

How many cycles the processor needs to stall before we figure out the next instruction after “bne”?

A. 0  
B. 1  
C. 2  
D. 3  
E. 4