Processor Design – Pipelined Processor

Hung-Wei Tseng
Outline

• Pipelining
• Designing a 5-stage pipeline processor for MIPS
Pipelining
Pipelining

- Break up the logic with “pipeline registers” into pipeline stages
- Each stage can act on different instruction/data
- States/Control signals of instructions are held in pipeline registers
After the 5th cycle, the processor can do 5 instructions in parallel
Pipelining

The processor can complete 1 instruction each cycle

CPI == 1 if everything works perfectly!
Single-cycle v.s. pipeline v.s.
Cycle time of a pipeline processor

- Critical path is the longest possible delay between two registers in a design.
- The critical path sets the cycle time, since the cycle time must be long enough for a signal to traverse the critical path.
- Lengthening or shortening non-critical paths does not change performance.
- Ideally, all paths are about the same length.
Designing a 5-stage pipeline processor for MIPS
Basic steps of execution

- Instruction fetch: fetch an instruction from memory
- Decode:
  - What's the instruction?
  - Where are the operands?
- Execute
- Memory access
  - Where is my data? (The data memory address)
- Write back
  - Where to put the result
- Determine the next PC
Pipeline a MIPS processor

- **Instruction Fetch**
  - Read the instruction

- **Decode**
  - Figure out the incoming instruction?
  - Fetch the operands from the register file

- **Execution: ALU**
  - Perform ALU functions

- **Memory access**
  - Read/write data memory

- **Write back results to registers**
  - Write to register file

---

**Instructions:**

- Instruction Fetch (IF)
- Instruction Decode (ID)
- Execution (EXE)
- Memory Access (MEM)
- Write Back (WB)
From single-cycle to pipeline

Instruction Fetch

Instruction Decode

Execution

Memory Access

Write Back

PCSrc = Branch & Zero

IF/ID

ID/EX

EX/MEM

MEM/WB

Instruction Memory

Register File

Data Memory

Add

Control

Write

Read

Address

ALU

Instruction Fetch

Instruction Decode

Execution

Memory Access

Write Back

Will this work?
Pipelined processor

add $1, $2, $3
lw  $4, 0($5)
sub $6, $7, $8
sub $9,$10,$11
sw  $1, 0($12)
Pipelined processor

```
add $1, $2, $3
lw  $4, 0($5)
sub $6, $7, $8
sub $9,$10,$11
sw  $1, 0($12)
```
Pipelined processor

Where can I find these?

add $1, $2, $3
lw $4, 0($5)
sub $6, $7, $8
sub $9,$10,$11
sw $1, 0($12)
Pipelined processor

```
add $1, $2, $3
lw  $4, 0($5)
sub $6, $7, $8
sub $9,$10,$11
sw  $1, 0($12)
```
Pipelined processor

Is this right?

```
add $1, $2, $3
lw  $4, 0($5)
sub $6, $7, $8
sub $9,$10,$11
sw  $1, 0($12)
```
Pipelined processor
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)
Pipeline hazards
Pipeline hazards

- Even though we perfectly divide pipeline stages, it’s still hard to achieve 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
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)
```
Structural hazard

- The design of hardware cause structural hazard
- We need to modify the hardware design to avoid structural hazard
Data hazard
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
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
Performance of stall

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

15 cycles! CPI == 3
(If there is no stall, CPI should be just 1!)
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 obtain the result here!
Sol. of data hazard II: Forwarding

- Take the values, where ever they are!

```
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!)
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
Forwarding in hardware
There is still a case that we have to stall...

- Revisit the following code:

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

- 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 in hardware
Control hazard
Control hazard

- The processor cannot determine the next PC to fetch

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

7 cycles per loop
Solution 1: Delayed branches

• An agreement between ISA and hardware
  • “Branch delay” slots: the next N instructions after a branch are always executed
  • Compiler decides the instructions in branch delay slots
    • Reordering the instruction cannot affect the correctness of the program
    • MIPS has one branch delay slot

• Good
  • Simple hardware

• Bad
  • N cannot change
  • Sometimes cannot find good candidates for the slot
Solution I: Delayed branches

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

branch delay slot

```
6 cycles per loop
```
Solution II: always predict not-taken

- Always predict the next PC is PC+4

```
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)  
      add $t4, $t3, $t5  
      lw $t3, 0($s0)
```

If branch is not taken: no stalls!
If branch is taken: no hurt!

7 cycles per loop

flush the instructions fetched incorrectly
Solution III: always predict taken
Solution III: always predict taken

Still have to stall 1 cycle
Solution III: always predict taken

Consult BTB in fetch stage
Branch Target Buffer

PC

branch PC

<table>
<thead>
<tr>
<th>target address or target instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Solution III: always predict taken

• Always predict taken with the help of BTB

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

5 cycles per loop
(CPI == 1 !!!)

But what if the branch is not always taken?
Dynamic branch prediction
1-bit counter

- Predict this branch will go the same way as the result of the last time this branch executed
- 1 for taken, 0 for not taken

PC = 0x400420

Branch Target Buffer

<table>
<thead>
<tr>
<th>PC</th>
<th>Target Address</th>
<th>Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400420</td>
<td>0x8048324</td>
<td>1</td>
</tr>
<tr>
<td>0x400464</td>
<td>0x8048392</td>
<td>1</td>
</tr>
<tr>
<td>0x400578</td>
<td>0x804850a</td>
<td>0</td>
</tr>
<tr>
<td>0x41000C</td>
<td>0x8049624</td>
<td>1</td>
</tr>
</tbody>
</table>

Taken!
2-bit counter

- A 2-bit counter for each branch
- If the prediction in taken states, fetch from target PC, otherwise, use PC+4

<table>
<thead>
<tr>
<th>Taken (11)</th>
<th>Taken (10)</th>
<th>Not Taken (00)</th>
<th>Not Taken (01)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Taken</th>
<th>Not Taken</th>
</tr>
</thead>
</table>

Branch Target Buffer

<table>
<thead>
<tr>
<th>PC</th>
<th>Target Address</th>
<th>Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400420</td>
<td>0x8048324</td>
<td>11</td>
</tr>
<tr>
<td>0x400464</td>
<td>0x8048392</td>
<td>10</td>
</tr>
<tr>
<td>0x400578</td>
<td>0x804850a</td>
<td>00</td>
</tr>
<tr>
<td>0x41000C</td>
<td>0x8049624</td>
<td>01</td>
</tr>
</tbody>
</table>

PC = 0x400420

Taken!
Performance of 2-bit counter

• 2-bit state machine for each branch

```
for(i = 0; i < 10; i++) {
    sum += a[i];
}
```

90% accuracy!

<table>
<thead>
<tr>
<th>i</th>
<th>state</th>
<th>predict</th>
<th>actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>4-9</td>
<td>11</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>T</td>
<td>NT</td>
</tr>
</tbody>
</table>

• Application: 80% ALU, 20% Branch, and branch resolved in EX stage, average CPI?

• \( 1 + 20\% \times (1 - 90\%) \times 2 = 1.04 \)
Make the prediction better

- Consider the following code:

```c
i = 0;
do {
    if (i % 3 != 0) // Branch Y, 
        taken if i % 3 == 0
        a[i] *= 2;
    a[i] += i;
} while (++i < 100) // Branch X
```

Can we capture the pattern?
Predict using history

• Instead of using the PC to choose the predictor, use a bit vector (global history register, GHR) made up of the previous branch outcomes.

• Each entry in the history table has its own counter.

\[ n\text{-bit GHR} = 101 \text{ (T, NT, T)} \]
Performance of global history predictor

• Consider the following code:

```c
i = 0;
do {
    if( i % 3 != 0) // Branch Y, taken if i % 3 == 0
        a[i] *= 2;
    a[i] += i;
    // Branch Y
} while ( ++i < 100) // Branch X
```

Assume that we start with a 4-bit GHR= 0, all counters are 10.

![Table showing branch history and prediction](image)

Nearly perfect after this
Deep pipeline and branch prediction
Pentium 4

- Very deep pipeline: in order to achieve high frequency! (start from 1.5GHz)
  - 20 stages in Netburst
  - 31 stages in Prescott
  - 103W (3.6GHz, 65nm)

Reference
- The Microarchitecture of the Pentium 4 Processor
Athlon 64

- 12 stage pipeline

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Inst. Addr Decode</td>
</tr>
<tr>
<td>2</td>
<td>Inst Mem Read</td>
</tr>
<tr>
<td>3</td>
<td>Inst. Byte Pick</td>
</tr>
<tr>
<td>4</td>
<td>ID1</td>
</tr>
<tr>
<td>5</td>
<td>ID2</td>
</tr>
<tr>
<td>6</td>
<td>Inst. Dbl. &amp; Pack</td>
</tr>
<tr>
<td>7</td>
<td>ID and Pack</td>
</tr>
<tr>
<td>8</td>
<td>Dispatch</td>
</tr>
<tr>
<td>9</td>
<td>Scheduling</td>
</tr>
<tr>
<td>10</td>
<td>Execution</td>
</tr>
<tr>
<td>11</td>
<td>D-Cache Address</td>
</tr>
<tr>
<td>12</td>
<td>D-cache Access</td>
</tr>
</tbody>
</table>

- 89W TDP (Opteron 2.2GHz 90nm)
Data hazard revisited

• How many cycles it takes to execute the following code?
• Draw the pipeline execution diagram
  • assume that we have full data forwarding.

```
lw   $t1, 0($a0)  
lw   $a0, 0($t1)  
  bne  $a0, $zero, 0
```

9 cycles
Case Study
Pentium 4 Microarch.

[Diagram showing the architecture of the Pentium 4 processor, including components such as Front-End BTB (4K Entries), Instruction TLB/Prefetcher, Instruction Decoder, Trace Cache BTB (512 Entries), Trace Cache (12K µops), Allocator/Register Renamer, Memory uop Queue, Memory Scheduler, Integer/Floating Point uop Queue, Fast, Slow/General FP Scheduler, Simple FP, Integer Register File/Bypass Network, FP Register/Bypass, AGU (_Load Address, Store Address), 2x ALU (Simple Instr.), Slow ALU (Complex Instr.), FP MMX, SSE, SSE2, FP Move, L2 Cache (256K Byte 8-way), 48GB/s, 64-bits wide, System Bus, 256 bits, L1 Data Cache (8Kbyte 4-way).]
Athlon 64

AMD K8 Architecture

[Diagram of Athlon 64 architecture with various components and connections]
Demo revisited

• Why the sorting the array speed up the code by 2.75x despite the increased instruction count?

```cpp
if (option)
    std::sort(data, data + arraySize);

for (unsigned i = 0; i < 100000; ++i) {
    for (unsigned c = 0; c < arraySize; ++c) {
        if (data[c] >= 128)
            sum += data[c];
    }
}
```
Q & A