Processor Design — Pipelined Processor

Hung-Wei Tseng
Drawbacks of a single-cycle processor

- The cycle time is determined by the longest instruction
  - Could be very long, thinking about fetch data from DRAM
- Hardware is mostly idle
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 hold in pipeline registers (latches)
Pipelining

- **cycle #1**
  - 2ns latch latch latch latch latch latch

- **cycle #2**
  - 2ns latch latch latch latch latch latch

- **cycle #3**
  - 2ns latch latch latch latch latch latch

- **cycle #4**
  - 2ns latch latch latch latch latch latch

- **cycle #5**
  - 2ns latch latch latch latch latch latch
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.
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

Diagram:

- Instruction Fetch (IF)
- Instruction Decode (ID)
- Execution (EXE)
- Memory Access (MEM)
- Write Back (WB)
Pipelined datapath

Instruction Fetch
Instruction Decode
Execution
Memory Access
Write Back

Instruction Memory
Read Address
inst[31:0]

Add
4

ID/EX
RegWrite

Shift left 2

EX/MEM
ALUop

MEM/WB
MemWrite

IF/ID
PCCsrc = Branch & Zero

Data Memory
Read Data
Write Data

RegDst

MEMtoReg

ALUSrc
Zero

address

Will this work?
add $1, $2, $3
lw  $4, 0($5)
sub $6, $7, $8
sub $9,$10,$11
sw  $1, 0($12)
Pipelined datapath

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

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

Is this right?

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

Instruction Memory
Read Address inst[31:0]

IF/ID
PCSrc
Add
4

Register
RegWrite
inst[25:21]

ID/EX
Read Reg 1
Read Reg 2
Read Data 1
Write Reg
Write Data

EX/MEM
Shift left 2
Add
MemWrite
MemRead

MEM/WB
Data Memory
RegDst
ALUop
ALUSrc
Write Data
MemtoReg
Address
Read Data
Pipelined datapath + control
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!

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

PC → Instruction Memory

IF/ID Stage:
- Instruction Address
- Instruction Fetch

ID/EX Stage:
- Instruction Decode
- Register Read
- Data Read
- ALU Operation
- Forwarding Unit

EX/MEM Stage:
- ALU Result
- Memory Access
- Register Write

MEM/WB Stage:
- Memory Write
- Register Dst Write
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 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)
Control hazard
Control hazard

- The processor cannot determine the next PC to fetch

```
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)
```

7 cycles per loop
Solution I: 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

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

branch delay slot

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

- Always predict the next PC is PC+4

```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)
        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
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

Branch PC  target address or target instruction

Branch Target Buffer
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

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</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>

Branch Target Buffer

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>PC</th>
<th>Target Address</th>
<th>Branch Taken</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>

Branch Target Buffer

PC = 0x400420

Taken!
Performance of 2-bit counter

• 2-bit state machine for each branch

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

90% prediction rate!

• 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.

\[ \text{n-bit GHR} = 101 \text{ (T, NT, T)} \]

\[ 2^n \text{ entries} \]

Taken!
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;
} while ( ++i < 100) // Branch X
```

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

<table>
<thead>
<tr>
<th>i</th>
<th>?</th>
<th>GHR</th>
<th>BHT</th>
<th>prediction</th>
<th>actual</th>
<th>New BHT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Y</td>
<td>0000</td>
<td>10</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>0001</td>
<td>10</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>Y</td>
<td>0011</td>
<td>10</td>
<td>T</td>
<td>NT</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>0110</td>
<td>10</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>Y</td>
<td>1101</td>
<td>10</td>
<td>T</td>
<td>NT</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>1010</td>
<td>10</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
<td>0101</td>
<td>10</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>1011</td>
<td>10</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>Y</td>
<td>0111</td>
<td>10</td>
<td>T</td>
<td>NT</td>
<td>01</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td>1110</td>
<td>10</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>Y</td>
<td>1101</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
<td>00</td>
</tr>
<tr>
<td>5</td>
<td>X</td>
<td>1010</td>
<td>11</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>Y</td>
<td>0101</td>
<td>11</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>X</td>
<td>1011</td>
<td>11</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>Y</td>
<td>0111</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
<td>00</td>
</tr>
<tr>
<td>7</td>
<td>X</td>
<td>1110</td>
<td>11</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>Y</td>
<td>1101</td>
<td>00</td>
<td>NT</td>
<td>NT</td>
<td>00</td>
</tr>
<tr>
<td>8</td>
<td>X</td>
<td>1010</td>
<td>11</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>Y</td>
<td>0101</td>
<td>11</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>X</td>
<td>1011</td>
<td>11</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>Y</td>
<td>0111</td>
<td>00</td>
<td>NT</td>
<td>NT</td>
<td>00</td>
</tr>
</tbody>
</table>

Nearly perfect after this