Midterm Review

Hung-Wei Tseng
Today’s von Neumann computers

- Instruction fetch: where? instruction memory
- Decode:
  - What’s the instruction?
  - Where are the operands? registers
- Execute ALUs
- Memory access
  - Where is my data? data memory
- Write back registers
  - Where to put the result
- Determine the next PC
ISA
Instruction Set Architecture (ISA)

- The contract between the hardware and software
- Defines the set of operations that a computer/processor can execute
- Programs are combinations of these instructions
  - Abstraction to programmers/compilers
- The hardware implements these instructions in any way it choose.
  - Directly in hardware circuit. e.g. CPU
  - Software virtual machine. e.g. VirtualPC
  - Simulator/Emulator. e.g. DeSmuME
  - Trained monkey with pen and paper
MIPS ISA

- All instructions are 32 bits
- 32 32-bit registers
  - All registers are the same
  - $zero is always 0
- 50 opcodes

<table>
<thead>
<tr>
<th>name</th>
<th>number</th>
<th>usage</th>
<th>saved?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>zero</td>
<td>N/A</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>assembler temporary</td>
<td>no</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>return value</td>
<td>no</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>arguments</td>
<td>no</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>saved</td>
<td>yes</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
<td>yes</td>
</tr>
</tbody>
</table>
MIPS Instruction “formats”

• 3 instruction formats
  • R-type: all operands are registers
    6 bits  5 bits  5 bits  5 bits  5 bits  6 bits
    opcode rs rt rd shift amount funct
  • I-type: one of the operands is an immediate value
    6 bits  5 bits  5 bits  16 bits
    opcode rs rt immediate / offset
  • J-type: non-conditional, non-relative branches
    6 bits  26 bits
    opcode target
Categorize MIPS instructions according to their “functions”

- **ALU instructions**: perform ALU operations, put the execution result in the destination register
  - add, sub, and, xor
  - addi, subi, andi

- **Memory instructions**: access memory
  - lw: load data from memory to the destination register
  - sw: store data from a register to destination memory address

- **Branch instructions**: change the next PC according to the comparison result
  - beq, bne

- **Jump**: change the PC
Practice

• Translate the C code into assembly:

```c
for(i = 0; i < 100; i++)
{
    sum+=A[i];
}
```

```assembly
Assume int is 32 bits
$s0 = &A[0]$
$v0 = sum;
$t0 = i;$

1. Initialization
2. Load A[i] from memory to register
3. Add the value of A[i] to sum
4. Increase by 1
5. Check if i still < 100

There are many ways to translate the C code. But efficiency may be differ among translations.
Function calls

**Caller**

- addi $a0, $t1, $t0
- sll $v0, $v0, 1
- addi $v0, $v0, 1
- li $v0, 4
- syscall

**Callee**

```
addi $a0, $t1, $t0
jal hanoi
sll $v0, $v0, 1
addi $v0, $v0, 1
t0
```

```
PC1:
jal hanoi
sll $v0, $v0, 1
addi $v0, $v0, 1
hanoi_0:
addi $a0, $a0, -1
bne $a0, $zero, hanoi_1
addi $v0, $zero, 1
j return
hanoi_1:
jal hanoi
sll $v0, $v0, 1
addi $v0, $v0, 1
```

**Function calls**

- **Callee**
  - hanoi:
    - addi $sp, $sp, -8
    - sw $ra, 0($sp)
    - sw $a0, 4($sp)
  - hanoi_0:
    - addi $a0, $a0, -1
    - bne $a0, $zero, hanoi_1
    - addi $v0, $zero, 1
    - j return
  - hanoi_1:
    - jal hanoi
    - sll $v0, $v0, 1
    - addi $v0, $v0, 1
- **Caller**
  - addi $a0, $t1, $t0
  - sll $v0, $v0, 1
  - addi $v0, $v0, 1
  - li $v0, 4
  - syscall

**save shared registers to the stack, maintain the stack pointer**

**restore shared registers from the stack, maintain the stack pointer**
Demo

- The overhead of function calls
- The keyword `inline` in C can embed the callee code at the call site
  - Eliminates function call overhead
- Does not work if it’s called using a function pointer
x86

- The most widely used ISA
- A poorly-designed ISA
  - It breaks almost every rule of a good ISA
    - variable length of instructions
    - the work of each instruction is not equal
    - makes the hardware become very complex
  - It’s popular != It’s good
- You don’t have to know how to write it, but you need to be able to read them and compare x86 with other ISAs
- Reference
Summation for x86

- Translate the C code into assembly:

```c
for(i = 0; i < 100; i++)
{
    sum+=A[i];
}
```

```
xorl %eax, %eax
.L2: addl (%ecx,%eax,4), %edx
    addl $1, %eax
    cmpl $100, %eax
    jne .L2
```

Assume
int is 32 bytes
%ecx = &A[0]
%edx = sum;
%eax = i;
# MIPS v.s. x86

<table>
<thead>
<tr>
<th></th>
<th>MIPS</th>
<th>x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISA type</td>
<td>RISC</td>
<td>CISC</td>
</tr>
<tr>
<td>instruction width</td>
<td>32 bits</td>
<td>1 ~ 17 bytes</td>
</tr>
<tr>
<td>code size</td>
<td>larger</td>
<td>smaller</td>
</tr>
<tr>
<td>registers</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>addressing modes</td>
<td>reg+offset</td>
<td>base+offset</td>
</tr>
<tr>
<td></td>
<td></td>
<td>base+index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scaled+index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scaled+index+offset</td>
</tr>
<tr>
<td>hardware</td>
<td>simple</td>
<td>complex</td>
</tr>
</tbody>
</table>
Performance
Performance Equation

Execution Time = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}

- How many instruction executed?
- How long is it to execute each instruction

- ET = IC \times CPI \times CT
- IC (Instruction Count)
- CPI (Cycles Per Instruction)
- CT (Seconds Per Cycle)
- 1 Hz = 1 second per cycle; 1 GHz = 1 ns per cycle
Summary: Performance Equation

Execution Time = \( \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}} \)

- ET = IC * CPI * Cycle Time
- IC (Instruction Count)
  - ISA, Compiler, algorithm, programming language, programmer
- CPI (Cycles Per Instruction)
  - Machine Implementation, microarchitecture, compiler, application, algorithm, programming language, programmer
- Cycle Time (Seconds Per Cycle)
  - Process Technology, microarchitecture, programmer
Speedup

• Compare the relative performance of the baseline system and the improved system
• Definition

\[
\text{Speedup} = \frac{\text{Execution time}_{\text{baseline}}}{\text{Execution time}_{\text{improved system}}}
\]
Amdahl’s Law

Speedup = \frac{1}{((\frac{x}{S})+(1-x))}

- \( x \): the fraction of “execution time” that we can speed up in the target application
- \( S \): by how many times we can speed up \( x \)

total execution time = 1

new execution time = \((\frac{x}{S})+(1-x))"
Amdahl’s Corollary #1

- Maximum possible speedup $S_{\text{max}}$, if we are targeting $x$ of the program.

$$S = \text{infinity}$$

$$S_{\text{max}} = \frac{1}{\left(\frac{x}{\text{inf}} + (1-x)\right)}$$

$$S_{\text{max}} = \frac{1}{(1-x)}$$
Amdahl’s Corollary #2

- Make the **common case** fast (i.e., x should be large)!
- Common == most time consuming not necessarily “most frequent”
- The uncommon case doesn’t make much difference
- Be sure of what the common case is
- The common case can change based on inputs, compiler options, optimizations you’ve applied, etc.
Case study: LOL

- Corollary #2
- The CPU is not the main performance bottleneck
- CPU parallelism doesn’t help, either
- You might consider
  - GPU
  - network
  - storage (loading maps)
Amdahl’s Corollary #3

• Assume that we have an application, in which $x$ of the execution time in this application can be fully parallelized with $S$ processors. What’s the speedup if we use a $S$-core processor instead of a single-core processor?

$$S_{par} = \frac{1}{\frac{x}{S} + (1-x)}$$
Multiple optimizations

- We can apply Amdahl’s law for multiple optimizations
- These optimizations must be dis-joint!
  - If optimization #1 and optimization #2 are dis-joint:
    \[
    \text{Speedup} = \frac{1}{(1 - X_{\text{Opt1}} - X_{\text{Opt2}}) + \frac{X_{\text{Opt1}}}{S_{\text{Opt1}}} + \frac{X_{\text{Opt2}}}{S_{\text{Opt2}}}}
    \]
  - If optimization #1 and optimization #2 are not dis-joint:
    \[
    S = \frac{1}{(1 - X_{\text{Opt1\ Only}} - X_{\text{Opt2\ Only}} - X_{\text{Opt1\&\ Opt2}}) + \frac{X_{\text{Opt1}}}{S_{\text{Opt1\ Only}}} + \frac{X_{\text{Opt2}}}{S_{\text{Opt2\ Only}}} + \frac{X_{\text{Opt1\&\ Opt2}}}{S_{\text{Opt1\&\ Opt2}}}}
    \]

Total execution time = 1
Is GFLOPS (Giga FLoating-point Operations Per Second) a good metric?

\[
\text{GFLOPS} = \frac{\text{# of floating point instructions}}{10^9} \times \frac{1}{\text{Execution Time}}
\]

\[
= \frac{\text{IC} \times \% \text{ of floating point instructions}}{\text{IC} \times \text{CPI} \times \text{CycleTime} \times 10^9} = \frac{\text{Clock Rate} \times \% \text{ FP ins.}}{\text{CPI} \times 10^9}
\]

• Cannot compare different ISA/compiler
  • What if the compiler can generate code with fewer instructions?
  • What if new architecture has more IC but also lower CPI?
• Does not make sense if the application is not floating point intensive
Power & Energy
Power

• Dynamic power: \( P = aCV^2f \)
  • \( a \): switches per cycle
  • \( C \): capacitance
  • \( V \): voltage
  • \( f \): frequency, usually linear with \( V \)
  • Doubling the clock rate consumes more power than a quad-core processor!

• Static/Leakage power becomes the dominant factor in the most advanced process technologies.

• Power is the direct contributor of “heat”
  • Packaging of the chip
  • Heat dissipation cost
Energy

- Energy = P * ET
- The electricity bill and battery life is related to energy!
- Lower power does not necessarily mean better battery life if the processor slows down the application too much.
Processor Design -- Single-cycle processor
Implementing an R-type instruction

- **opcode**: 6 bits
- **rs**: 5 bits
- **rt**: 5 bits
- **rd**: 5 bits
- **shift amt**: 5 bits
- **funct**: 6 bits

 instruction = MEM[PC]
 REG[rd] = REG[rs] op REG[rt]
 PC = PC + 4

Tell the processor when to start an instruction

Tell the ALU what ALU function to perform
Implementing a load instruction

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate / offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

Instruction = MEM[PC]
REG[rt] = MEM[signext(immediate) + REG[rs]]
PC = PC + 4
Implementing a store instruction

\[
\text{instruction} = \text{MEM}[\text{PC}]
\]
\[
\text{MEM}[\text{signext}(\text{immediate}) + \text{REG}[\text{rs}]] = \text{REG}[\text{rt}]
\]
\[
\text{PC} = \text{PC} + 4
\]
Implementing a branch instruction

<table>
<thead>
<tr>
<th></th>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>16 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td>immediate / offset</td>
<td></td>
</tr>
</tbody>
</table>

instruction = MEM[PC]
PC = (REG[rs] == REG[rt]) ? PC + 4 + SignExtImmediate *4 : PC + 4

Calculate the target address
Recap: 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
Processor Design -- Pipelined processor
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
The processor can complete 1 instruction each cycle

CPI == 1 if everything works perfectly!
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.
From single-cycle to pipeline

Instruction Fetch
PCSrc = Branch & Zero
PCSrc

Instruction Decode

Execution
RegWrite
Shift left 2
ALU
ALUop
Zero

Memory Access

Write Back

IF/ID
Add
Instruction Memory
Read Address
inst[31:0]

ID/EX

EX/MEM

MEM/WB

Data Memory
ALUSrc
Write Data

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

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
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
Can we get the right result?

- Given the current 5-stage pipeline, how many of the following MIPS code can work correctly?

<table>
<thead>
<tr>
<th></th>
<th>IF</th>
<th>ID</th>
<th>EXE</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>add $1, $2, $3</td>
<td>lw $4, 0($1)</td>
<td>sub $6, $7, $8</td>
<td>sw $1, 0($12)</td>
<td></td>
</tr>
<tr>
<td>II</td>
<td>add $1, $2, $3</td>
<td>lw $4, 0($5)</td>
<td>sub $6, $7, $8</td>
<td>sw $1, 0($12)</td>
<td></td>
</tr>
<tr>
<td>III</td>
<td>add $1, $2, $3</td>
<td>lw $4, 0($5)</td>
<td>bne $0, $7, L</td>
<td>sw $1, 0($12)</td>
<td></td>
</tr>
<tr>
<td>IV</td>
<td>add $1, $2, $3</td>
<td>lw $4, 0($5)</td>
<td>sub $9, $10, $11</td>
<td>sw $1, 0($12)</td>
<td></td>
</tr>
</tbody>
</table>

- b cannot get $1 produced by a before WB
- both a and d are accessing $1 at 5th cycle
- We don’t know if d & e will be executed or not

Data hazard | Structural hazard | Control hazard
Pipelined processor -- Structural hazards
Structural hazard

- The hardware cannot support the combination of instructions that we want to execute at the same cycle.
- We need to modify the hardware to solve this hazard.
- The original pipeline incurs structural hazard when two instructions competing the same register.
  - 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)
```
Pipelined processor --
Data hazards
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

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!
- Forward from the output of a pipeline register to functional units of the depending instructions.

```
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!)

Iw generates result at MEM stage, we have to stall
Pipelined processor --
Control hazards
Control hazard

- The processor cannot determine the next PC to fetch until the branch resolved in **EXE** stage

```plaintext
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

you don’t need to draw this
Mitigating control hazards

- Static approaches
  - Delayed branch
  - Always predict not-taken
  - Always predict taken

- Dynamic branch predictions
  - Local predictors: counters associated with each branch PC
    - 1-bit counters associated with each branch PC in BTB
    - 2-bit counters associated with each branch PC in BTB
  - Global predictor
    - 2-bit counters associated with each “history pattern”
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 program correctness
- MIPS has one branch delay slot

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

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

LOOP: lw $t3, 0($s0)
```

If branch is not taken: no stalls!
If branch is taken: doesn’t hurt!

7 cycles per loop
Solution III: always predict taken

Consult BTB in fetch stage
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

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?
Local 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!
Accuracy of 1-bit counter

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
```

What is the prediction accuracy of branch Y using 1-bit predictors (if all counters start with 0/not taken). Choose the most close one.

Assume unlimited BTB entries.

<table>
<thead>
<tr>
<th>i</th>
<th>branch</th>
<th>predict</th>
<th>actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Y</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>Y</td>
<td>T</td>
<td>NT</td>
</tr>
<tr>
<td>2</td>
<td>Y</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>Y</td>
<td>T</td>
<td>NT</td>
</tr>
<tr>
<td>5</td>
<td>Y</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>6</td>
<td>Y</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>7</td>
<td>Y</td>
<td>T</td>
<td>NT</td>
</tr>
</tbody>
</table>

A. 0%
B. 33%
C. 67%
D. 100%
Local 2-bit counter

• A 2-bit counter for each branch
• Predict taken if the counter value >= 2
• If the prediction in taken states, fetch from target PC, otherwise, use PC+4

- Taken 3 (11)
- Taken 2 (10)
- Not Taken 0 (00)
- Not Taken 1 (01)

Branch Target Buffer

<table>
<thead>
<tr>
<th>PC</th>
<th>Branch Target</th>
<th>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>

PC = 0x400420

Taken!
Accuracy of 2-bit counter

• 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
```

What is the prediction accuracy of branch Y using 2-bit predictors (if all counters start with 00). Choose the closest one. Assume unlimited BTB entries.

A. 0%
B. 33%
C. 67%
D. 100%

<table>
<thead>
<tr>
<th>i</th>
<th>branch</th>
<th>state</th>
<th>predict</th>
<th>actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>Y</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>2</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>Y</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>5</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>6</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>7</td>
<td>Y</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
</tbody>
</table>
Make the prediction better

- Consider the following code:

  ```
  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.

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

index

2^n entries

history table

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;
    // Branch Y
} 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
Deeper pipeline

- Higher frequencies by shortening the pipeline stages
- Higher marketing values since consumers usually link performance with frequencies
- Potentially higher power consumption as dynamic/active power = $aCV^2f$
- If the execution time is better, still consume less energy
Pentium 4 v.s. Athlon 64

- Application: 80% ALU, 20% Branch, 90% prediction accuracy, consider the two machines:
  - Pentium 4 with 20 pipeline stages, branch resolved in stage 19, running at 3 GHz
  - Athlon 64 with 12 pipeline stages, branch resolved in stage 10, running at 2.7 GHz (11% longer cycle time)

which one is faster?

\[
\text{CPI}_{\text{P4}} = 80\% \times 1 + 20\% \times 90\% \times 1 + 20\% \times 10\% \times 19 = 1.36 \\
\text{CPI}_{\text{Athlon64}} = 80\% \times 1 + 20\% \times 90\% \times 1 + 20\% \times 10\% \times 10 = 1.18 \\
\]

At least 15% faster clock rate to achieve the same performance
Tips of drawing pipeline diagram

• Each instruction has to go through all 5 pipeline stages: IF, ID, EXE, MEM, WB in order
• An instruction can enter the next pipeline stage in the next cycle if
  • No other instruction is occupying the next stage
  • This instruction has completed its own work in the current stage
  • The next stage has all its inputs ready
• Fetch a new instruction only if
  • We know the next PC to fetch
  • We can predict the next PC
  • Flush an instruction if the branch resolution says it’s mis-predicted.
Tips of drawing pipeline diagram

- Each instruction has to go through all 5 pipeline stages: IF, ID, EXE, MEM, WB in order
- An instruction can enter the next pipeline stage in the next cycle if
  - No other instruction is occupying the next stage
  - This instruction has completed its own work in the current stage
  - The next stage has all its inputs ready
- Fetch a new instruction only if
  - We know the next PC to fetch
  - We can predict the next PC
  - Flush an instruction if the branch resolution says it’s mis-predicted.

Assume full data forwarding, predict always taken

```
addi $a1, $zero, 2
LOOP: lw $t1, 0($a0)
lw $a0, 0($t1)
addi $a1, $a1, -1
bne $a1, $zero, LOOP
add $v0, $zero, $a1
```

```
addi $a1, $zero, 2
lw $t1, 0($a0)
lw $a0, 0($t1)
```
For midterm

- No cheat sheet allowed
- No cheating allowed
- We will use scantron
- We will have some problems require you to write
- You may bring a calculator
- You should bring pen/pencil/eraser
Sample midterm
MIPS v.s. x86

• Which of the following is NOT correct about these two ISAs?
  A. x86 provides more instructions than MIPS
  B. x86 usually needs more instructions to express a program
  C. An x86 instruction may access memory for 3 times
  D. An x86 instruction may be shorter than a MIPS instruction
  E. An x86 instruction may be longer than a MIPS instruction
Identify the performance bottleneck

Why does an Intel Core i7 @ 3.5 GHz usually perform better than an Intel Core i5 @ 3.5 GHz or AMD FX-8350@4GHz?

A. Because the instruction count of the program are different
B. Because the clock rate of AMD FX is higher
C. Because the CPI of Core i7 is better
D. Because the clock rate of AMD FX is higher and CPI of Core i7 is better
E. None of the above

Sysbench 2014 from http://www.anandtech.com/
Performance of a single-cycle processor

• How many of the following statements about a single-cycle processor is correct?
  • The CPI of a single-cycle processor is always 1
  • If the single-cycle implements lw, sw, beq, and add instructions, the sw instruction determines the cycle time
  • Hardware elements are mostly idle during a cycle
  • We can always reduce the cycle time of a single-cycle processor by supporting fewer instructions

A. 0
B. 1
C. 2
D. 3
E. 4
Limitations of pipelining

• How many of the following descriptions about pipelining is correct?
  • You can always divide stages into short stages with latches
  • Pipeline registers incur overhead for each pipeline stage
  • The latency of executing an instruction in a pipeline processor is longer than a single-cycle processor
  • The throughput of a pipeline processor is usually better than a single-cycle processor
  • Pipelining a stage can always improve cycle time

A. 1
B. 2
C. 3
D. 4
E. 5
Data dependences

• How many pairs of data dependences are there in the following code?
  
  ```assembly
  add $1, $2, $3
  lw  $4, 0($1)
  sub $5, $2, $4
  sub $1, $3, $1
  sw  $1, 0($5)
  ```

  A. 1  
  B. 2  
  C. 3  
  D. 4  
  E. 5  

Static branch predictions

• How many of the following about static branch prediction method is correct?
  • Comparing with stalls, static branch prediction mechanisms are never doing worse in our current MIPS 5-stage pipeline
  • A static branch prediction mechanism never changes the prediction result during program execution
  • “Flush” occurs only after the processor detects an incorrect branch prediction
  • “Always predict taken” cannot fetch a taken instruction during the ID stage of the branch instruction without the help of BTB

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

• Assume that we have an application composed with a total of 500000 instructions, in which 20% of them are the load/store instructions with an average CPI of 6 cycles, and the rest instructions are integer instructions with average CPI of 1 cycle. If the processor runs at 1GHz, how long is the execution time?
Example of Amdahl’s Law

- Call of Duty Black Ops II loads a zombie map for 10 minutes on my current machine, and spends 20% of this time in integer instructions.
- How much faster must you make the integer unit to make the map loading 1 minute faster?
Amdahl’s Law for multicore processors

• Assume that we have an application, in which 50% of the application can be fully parallelized with 2 processors. Assuming 80% of the parallelized part can be further parallelized with 4 processors, what’s the speed up of the application running on a 4-core processor?
Example

- Draw the pipeline execution diagram
  
  ```
  LOOP: lw $t1, 0($a0)
    lw $a0, 0($t1)
    addi $a1, $a1, -1
    bne $a1, $zero, LOOP
    add $v0, $zero, $a1
  ```

- Assume that we have no data forwarding and no branch prediction
- Assume that we have full data forwarding and always predict taken.
- Assume that we split the MEM stage into M1 and M2, and the memory data is ready after M2. The processor still has full forwarding and always predict taken.
Dynamic branch prediction

• Consider the following code, which branch predictor (2-bit local, 2-bit global history with 4-bit GHR) works the best?

```c
for(i = 0; i < 10; i++) {
    for(j = 0; j < 4; j++) {
        sum+=a[i][j]
    }
}
```
Other things to think ...

• What is performance equation? What affects each term in the equation?
• What is Amdahl’s law?
• What is instruction set architecture?
• What is process of generating a binary from C source files?
• What are the architectural states of a program?
• What are the differences between MIPS and x86?
• What are the uniformity of MIPS?
• Why power consumption is an important issue in computer system design?
Other things to think ...

• Why GFLOPS (Giga FLoating-Point Operations Per Second) is not a proper performance metric in most cases?
• What are the drawbacks of a single cycle processor?
• What are the advantages of pipelining?
• What is clocking methodology?
• What are the basic steps of executing an instruction?
• What are pipeline hazards? Please explain and give examples
• How to solve the pipeline hazards?
• Code optimization demoed in class