# Quiz 4

## Question 1: Multiple Answer

Which of the following are throughput (or bandwidth) metrics?

<table>
<thead>
<tr>
<th>Correct Answers</th>
<th>Percent Correct</th>
<th>Percent Incorrect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle time</td>
<td>94.737%</td>
<td>5.263%</td>
</tr>
<tr>
<td>Instruction Count</td>
<td>96.491%</td>
<td>3.509%</td>
</tr>
<tr>
<td>Instructions/second</td>
<td>94.737%</td>
<td>5.263%</td>
</tr>
<tr>
<td>(Instructions/sec)/Watt</td>
<td>91.228%</td>
<td>8.772%</td>
</tr>
<tr>
<td>Frames per second.</td>
<td>87.719%</td>
<td>12.281%</td>
</tr>
</tbody>
</table>

Average Score: 1.47368 points
What is true about the following x86 instructions:
1. `pushl %eax`
2. `ret`
3. `enter`
4. `movl %eax, %ebx`
5. `movl %eax, -4(%ebx,%ecx,4)`

Correct Answers

- Only one of them can be translated into a single instructions of MIPS assembly.  
  Percent Correct: 75.439%  Percent Incorrect: 24.561%
- None of them would typically appear in highly-optimized code.  
  Percent Correct: 92.982%  Percent Incorrect: 7.018%
- All of the except #4 access memory.  
  Percent Correct: 70.175%  Percent Incorrect: 29.825%
- Only #1, #3, and #5 modify memory.  
  Percent Correct: 71.93%  Percent Incorrect: 28.07%
- Only #1, #2, and #5 modify memory
  All of them access memory.  
  Percent Correct: 89.474%  Percent Incorrect: 10.526%
- #1, #2, #3, #5 all involve arithmetic operations.  
  Percent Correct: 77.193%  Percent Incorrect: 22.807%
Consider my daily commute: My home is 1.4 miles from the freeway entrance, and there are 6.7 miles of freeway between the entrance and Genesee St, where I exit. My office is 1.6 miles from the freeway. The speed limit on non-freeways is 25MPH. On the freeway, it's 65MPH. I decide that I want to reduce my commute time by going twice as fast as is allowed by law, but I only want to speed on either the freeway or the non-freeway roads. According to Amdahl's law, which one should I choose if I want to save the most time?

Correct

- Speeding on the freeways.  
  33.333%

- Speeding on the non-freeways.  
  66.667%

- Speeding on either of them will give the same commute time.  
  0%

- I want to get this question wrong, so I marked this answer.  
  0%

Unanswered  
0%
Question 4: Multiple Choice

For a given set of resources (e.g., a network link, or a set of cashiers at the Sunshine store) and some work waiting to be done (e.g., information to be sent over the link, or customers to process), it is usually the case that

Correct

- If utilization is high, then latency is probably long. 70.175%
- If throughput is high, then latency is probably short. 21.053%
- There is no predictable relationship between latency and throughput or utilization. 8.772%

I love the Sunshine store, and I want to express that love more than I want to get this problem correct. 0%

Unanswered 0%
Question 5: Multiple Choice

Which instruction does x86 use to load a value from memory into a register.

Correct

- load
- get
- mov
- peek

Percent Answered

- load 0%
- get 0%
- mov 100%
- peek 0%

Unanswered
The Midterm is Coming

• Midterm on May 7th.
• Midterm review on May 2nd.
  • Come to class with questions.
• Midterm will cover everything before it
• It will mostly resemble the homeworks and quizzes
• It will be challenging.
• It will be curved.
Quiz 2

![Bar Chart]

- **F**: 22 students
- **F+**: 8 students
- **D-**: 5 students
- **D**: 5 students
- **D+**: 3 students
- **C-**: 2 students
- **C**: 1 student
- **C+**: 1 student
- **B-**: 1 student
- **B**: 1 student
- **B+**: 3 students
- **A-**: 1 student
- **A**: 2 students
- **A+**: 2 students

The chart shows the distribution of grades among the students for Quiz 2.
Quiz 3

The bar chart shows the distribution of grades among students:

- F: 5 students
- F+: 1 student
- D-: 2 students
- D: 1 student
- D+: 1 student
- C-: 3 students
- C: 2 students
- C+: 5 students
- B-: 7 students
- B: 12 students
- B+: 8 students
- A-: 6 students
- A: 6 students
- A+: 2 students

The y-axis represents the number of students, and the x-axis represents different grade categories.
Implementing a MIPS Processor

Readings: 4.1-4.11
Goals for this Class

- Understand how CPUs run programs
  - How do we express the computation the CPU?
  - How does the CPU execute it?
  - How does the CPU support other system components (e.g., the OS)?
  - What techniques and technologies are involved and how do they work?

- Understand why CPU performance (and other metrics) varies
  - How does CPU design impact performance?
  - What trade-offs are involved in designing a CPU?
  - How can we meaningfully measure and compare computer systems?

- Understand why program performance varies
  - How do program characteristics affect performance?
  - How can we improve a program's performance by considering the CPU running it?
  - How do other system components impact program performance?
Goals

• Understand how the 5-stage MIPS pipeline works
  • See examples of how architecture impacts ISA design
  • Understand how the pipeline affects performance

• Understand hazards and how to avoid them
  • Structural hazards
  • Data hazards
  • Control hazards
Processor Design in Two Acts

Act I: A single-cycle CPU
Foreshadowing

- **Act I: A Single-cycle Processor**
  - Simplest design – Not how many real machines work (maybe some deeply embedded processors)
  - Figure out the basic parts; what it takes to execute instructions

- **Act II: A Pipelined Processor**
  - This is how many real machines work
  - Exploit parallelism by executing multiple instructions at once.
Target ISA

- We will focus on part of MIPS
  - Enough to run into the interesting issues
  - Memory operations
  - A few arithmetic/Logical operations (Generalizing is straightforward)
  - BEQ and J
- This corresponds pretty directly to what you’ll be implementing in 141L.
Basic Steps for Execution

• Fetch an instruction from the instruction store
• Decode it
  • What does this instruction do?
• Gather inputs
  • From the register file
  • From memory
• Perform the operation
• Write back the outputs
  • To register file or memory
• Determine the next instruction to execute
The Processor Design Algorithm

• Once you have an ISA…
• Design/Draw the datapath
  • Identify and instantiate the hardware for your architectural state
  • Foreach instruction
    • Simulate the instruction
    • Add and connect the datapath elements it requires
    • Is it workable? If not, fix it.
• Design the control
  • Foreach instruction
    • Simulate the instruction
    • What control lines do you need?
    • How will you compute their value?
    • Modify control accordingly
    • Is it workable? If not, fix it.
• You’ve already done much of this in 141L.
• Arithmetic; R-Type
  • \text{Inst} = \text{Mem}[\text{PC}]
  • \text{REG}[\text{rd}] = \text{REG}[\text{rs}] \text{ op } \text{REG}[\text{rt}]
  • \text{PC} = \text{PC} + 4

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
</tr>
<tr>
<td># bits</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
Arithmetic; R-Type

- \( \text{Inst} = \text{Mem}[\text{PC}] \)
- \( \text{REG}[rd] = \text{REG}[rs] \text{ op } \text{REG}[rt] \)
- \( \text{PC} = \text{PC} + 4 \)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
</tr>
<tr>
<td># bits</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
- Arithmetic; R-Type
  - $\text{Inst} = \text{Mem}[\text{PC}]$
  - $\text{REG}[\text{rd}] = \text{REG}[\text{rs}] \text{ op } \text{REG}[\text{rt}]$
  - $\text{PC} = \text{PC} + 4$

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
</tr>
<tr>
<td># bits</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
• Arithmetic; R-Type
  • \( \text{Inst} = \text{Mem}[PC] \)
  • \( \text{REG}[rd] = \text{REG}[rs] \text{ op } \text{REG}[rt] \)
  • \( \text{PC} = \text{PC} + 4 \)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
</tr>
<tr>
<td># bits</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

![Diagram](image)
- **Arithmetic; R-Type**
  - $\text{Inst} = \text{Mem}[\text{PC}]$
  - $\text{REG}[\text{rd}] = \text{REG}[\text{rs}] \text{ op } \text{REG}[\text{rt}]$
  - $\text{PC} = \text{PC} + 4$

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
</tr>
<tr>
<td># bits</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
• **Arithmetic; R-Type**
  - \( \text{Inst} = \text{Mem}[\text{PC}] \)
  - \( \text{REG}[\text{rd}] = \text{REG}[\text{rs}] \ op \ \text{REG}[\text{rt}] \)
  - \( \text{PC} = \text{PC} + 4 \)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
</tr>
<tr>
<td># bits</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

**Diagram:**
- PC
- Instruction memory
- Instruction [31-0]
- Instruction [20-16]
- Instruction [25-21]
- Read address
- Read register 1
- Read register 2
- Write register
- Read data 1
- Read data 2
- Registers
- Jump Branch
- ALU
- ALU result
• Arithmetic; R-Type
  • Inst = Mem[PC]
  • REG[rd] = REG[rs] op REG[rt]
  • PC = PC + 4

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
</tr>
<tr>
<td># bits</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
• Arithmetic; R-Type
  • Inst = Mem[PC]
  • REG[rd] = REG[rs] op REG[rt]
  • **PC = PC + 4**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
</tr>
<tr>
<td># bits</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

```
4
Add

PC

Instruction memory

Instruction [31-0]

Instruction [25-21]

Instruction [20-16]

Instruction [11-15]

Read address

Read register 1

Read register 2

Read data 1

Read data 2

Write register

Write data

Registers

Jump Branch

ALU

ALU result
```
• ADDI; I-Type
  • $PC = PC + 4$
  • $REG[rt] = REG[rs] \text{ op } \text{SignExtImm}$

<table>
<thead>
<tr>
<th>bits</th>
<th>31:26</th>
<th>25:21</th>
<th>20:16</th>
<th>15:0</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>imm</td>
</tr>
<tr>
<td># bits</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>
• **ADDI; I-Type**
  
  - PC = PC + 4
  - REG[rt] = REG[rs] op \textbf{SignExtImm}

<table>
<thead>
<tr>
<th>bits</th>
<th>31:26</th>
<th>25:21</th>
<th>20:16</th>
<th>15:0</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>imm</td>
</tr>
<tr>
<td># bits</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>
• **Load Word**
  - **PC = PC + 4**
  - **REG[rt] = MEM[signextendImm + REG[rs]]**

<table>
<thead>
<tr>
<th>bits</th>
<th>31:26</th>
<th>25:21</th>
<th>20:16</th>
<th>15:0</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
</tr>
<tr>
<td># bits</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

Diagram of instruction processing with ALU operations and memory access.
**Load Word**

- \( PC = PC + 4 \)
- \( REG[rt] = MEM[\text{signextendImm + REG[rs]}] \)

<table>
<thead>
<tr>
<th>bits</th>
<th>31:26</th>
<th>25:21</th>
<th>20:16</th>
<th>15:0</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
</tr>
<tr>
<td># bits</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>
- **Store Word**
  - $PC = PC + 4$
  - $MEM[\text{signextend}\text{Imm} + \text{REG}[rs]] = \text{REG}[rt]$

<table>
<thead>
<tr>
<th>bits</th>
<th>31:26</th>
<th>25:21</th>
<th>20:16</th>
<th>15:0</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
</tr>
<tr>
<td># bits</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>
• **Store Word**
  - PC = PC + 4
  - \[\text{MEM}[^{\text{signextendImm} + \text{REG}[rs]}] = \text{REG}[rt]\]

<table>
<thead>
<tr>
<th>bits</th>
<th>31:26</th>
<th>25:21</th>
<th>20:16</th>
<th>15:0</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
</tr>
<tr>
<td># bits</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>
• Branch-equal; I-Type
  • \( PC = (\text{REG}[rs] == \text{REG}[rt]) \ ? \ PC + 4 + \text{SignExtImmediate} \times 4 : PC + 4; \)

<table>
<thead>
<tr>
<th>bits</th>
<th>31:26</th>
<th>25:21</th>
<th>20:16</th>
<th>15:0</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>displacement</td>
</tr>
<tr>
<td># bits</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>
• Branch-equal; I-Type
  - \( PC = (\text{REG}[rs] == \text{REG}[rt]) \) ? \( PC + 4 + \text{SignExtImmediate} \times 4 \) : \( PC + 4 \);

<table>
<thead>
<tr>
<th>bits</th>
<th>31:26</th>
<th>25:21</th>
<th>20:16</th>
<th>15:0</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>displacement</td>
</tr>
<tr>
<td># bits</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>
A Single-cycle Processor

• Performance refresher
• \( ET = IC \times CPI \times CT \)
• Single cycle \( \Rightarrow CPI = 1; \) That sounds great

• Unfortunately, Single cycle \( \Rightarrow CT \) is large
  • Even RISC instructions take quite a bite of effort to execute
  • This is a lot to do in one cycle
Our Hardware is Mostly Idle

Cycle time = 18 ns
Slowest module (alu) is ~6ns
Our Hardware is Mostly Idle

Cycle time = 18 ns
Slowest module (alu) is ~6ns
Processor Design in Two Acts

Act II: A pipelined CPU
Pipelining Review
Pipelining

What’s the latency for one unit of work?

What’s the throughput?
Pipelining

- Break up the logic with latches into “pipeline stages”
- Each stage can act on different data
- Latches hold the inputs to their stage
- Every clock cycle data transfers from one pipe stage to the next
What's the latency for one unit of work?

What's the throughput?
Critical path review

- 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.

Diagram:

```
  ( ) -> Logic -> ( )
```

31
Pipelining and Logic

• Hopefully, critical path is now 1/3 of what it was
Limitations of Pipelining

- You cannot pipeline forever
  - Some logic cannot be pipelined arbitrarily -- Memories
  - Some logic is inconvenient to pipeline.
  - How do you insert a register in the middle of a multiplier?

- Registers have a cost
  - They cost area -- choose “narrow points” in the logic
  - They cost energy -- latches don’t do any useful work
  - They cost time
    - Extra logic delay
    - Set-up and hold times.

- Pipelining may not affect the critical path as you expect
Pipelining Overhead

- Logic Delay (LD) -- How long does the logic take (i.e., the useful part)
- Set up time (ST) -- How long before the clock edge do the inputs to a register need be ready?
  - Relatively short -- 0.036 ns on our FPGAs
- Register delay (RD) -- Delay through the internals of the register.
  - Longer -- 1.5 ns for our FPGAs.
  - Much, much shorter for RAW CMOS.
Pipelining Overhead

Clock

Setup time

Register in

??????

New Data

Register out

Old data

New data

Register delay
Pipelining Overhead

- Logic Delay (LD) -- How long does the logic take (i.e., the useful part)
- Set up time (ST) -- How long before the clock edge do the inputs to a register need be ready?
- Register delay (RD) -- Delay through the internals of the register.
- $CT_{base}$ -- cycle time before pipelining
  - $CT_{base} = LD + ST + RD$.
- $CT_{pipe}$ -- cycle time after pipelining $N$ times
  - $CT_{pipe} = ST + RD + LD/N$
  - Total time = $N \times ST + N \times RD + LD$
• You can’t always pipeline how you would like
• The critical path only went down “fast logic”
How to pipeline a processor

• Break each instruction into pieces
  • We’ll base our break down on the basic algorithm for execution
    • Fetch
    • Decode
    • Collect arguments
    • Execute
    • Write back results
    • Compute next PC

• The “classic 5-stage MIPS pipeline”
  • Fetch -- read the instruction
  • Decode -- decode and read from the register file
  • Execute -- Perform arithmetic ops and address calculations
  • Memory -- access data memory.
  • Write back-- Store results in the register file.
Pipelining a processor
Pipelining a processor

Reality
Pipelining a processor

Easier to draw

Reality
Pipelining a processor

Reality

Easier to draw

Pipelined
Pipelined Datapath

Instruction Memory
- Read Address
- Read Addr 1
- Read Addr 2
- Write Addr
- Write Data

Register File
- Read Data 1
- Read Data 2

ALU
- Sign Extend
- 16
- 32

Data Memory
- Address
- Read Data
- Write Data

Memory
- Add

Read
- Address

Write
- Data
Pipelined Datapath

Instruction Memory

Read Address

Register

Read Addr 1
Write Addr
Write Data

Read Addr 2
File

Read Data 1
Read Data 2

ALU

Shift left 2
Add

Data Memory

Address
Read Data
Write Data

Sign Extend

add ...
lw ...
Subi...
Sub ....
Add ...
Add ...
Pipelined Datapath

Instruction Memory
- Read Address
- Read Addr 1
- Read Addr 2
- Write Addr
- Write Data

Register File
- Read Data 1
- Read Data 2

Dec/Exec
- Dec/Exec
- Dec/Exec
- Dec/Exec
- Dec/Exec

ALU
- Shlit
- Add
- Shift left 2

Data Memory
- Address
- Read Data
- Write Data

Mem/WB
- Sign
- Extend
- 16
- 32
This is a bug
rd must flow through
the pipeline with the instruction.
This signal needs to come from the WB stage.
This is a bug; rd must flow through the pipeline with the instruction. This signal needs to come from the WB stage.
Pipelined Control

• Control lives in decode stage, signals flow down the pipe with the instructions they correspond to.
Impact of Pipelining

- L = IC * CPI * CT
- Break the processor into P pipe stages
  - CT_{new} = CT/P
  - CPI_{new} = CPI_{old}
    - CPI is an average: Cycles/instructions
    - The latency of one instruction is P cycles
    - The average CPI = 1
  - IC_{new} = IC_{old}
- Total speedup should be 5x!
  - Except for the overhead of the pipeline registers
  - And the realities of logic design...
Pipelining Inaction
Pipelining Inaction
<table>
<thead>
<tr>
<th>Total</th>
<th>Incr</th>
<th>RF</th>
<th>Type</th>
<th>Fanout</th>
<th>Location</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.810</td>
<td>3.134</td>
<td>0.277</td>
<td>uTco</td>
<td>1</td>
<td>LCFF_X21_Y16_N25</td>
<td>inst_rom:rom</td>
</tr>
<tr>
<td>3.134</td>
<td>0.000</td>
<td>0.177</td>
<td>FF</td>
<td>CELL</td>
<td>23</td>
<td>LCFF_X21_Y16_N25</td>
</tr>
<tr>
<td>3.754</td>
<td>0.620</td>
<td>0.177</td>
<td>FF</td>
<td>IC</td>
<td>1</td>
<td>LCCOMB_X20_Y16_N20</td>
</tr>
<tr>
<td>3.931</td>
<td>0.588</td>
<td>0.177</td>
<td>FF</td>
<td>CELL</td>
<td>29</td>
<td>LCCOMB_X20_Y16_N20</td>
</tr>
<tr>
<td>4.877</td>
<td>0.946</td>
<td>0.178</td>
<td>RR</td>
<td>IC</td>
<td>1</td>
<td>LCCOMB_X19_Y12_N4</td>
</tr>
<tr>
<td>5.055</td>
<td>0.588</td>
<td>0.178</td>
<td>RR</td>
<td>CELL</td>
<td>5</td>
<td>LCCOMB_X19_Y12_N4</td>
</tr>
<tr>
<td>5.643</td>
<td>0.545</td>
<td>0.178</td>
<td>RR</td>
<td>CELL</td>
<td>1</td>
<td>LCCOMB_X18_Y12_N20</td>
</tr>
<tr>
<td>6.188</td>
<td>0.906</td>
<td>0.178</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB_X19_Y16_N30</td>
</tr>
<tr>
<td>7.094</td>
<td>0.959</td>
<td>0.178</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB_X19_Y16_N30</td>
</tr>
<tr>
<td>7.689</td>
<td>0.000</td>
<td>0.178</td>
<td>FF</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB_X19_Y15_N0</td>
</tr>
<tr>
<td>7.689</td>
<td>0.000</td>
<td>0.178</td>
<td>FR</td>
<td>CELL</td>
<td>1</td>
<td>LCCOMB_X19_Y15_N0</td>
</tr>
<tr>
<td>7.689</td>
<td>0.080</td>
<td>0.178</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB_X19_Y15_N2</td>
</tr>
<tr>
<td>7.849</td>
<td>0.080</td>
<td>0.178</td>
<td>RF</td>
<td>CELL</td>
<td>1</td>
<td>LCCOMB_X19_Y15_N2</td>
</tr>
<tr>
<td>7.849</td>
<td>0.000</td>
<td>0.178</td>
<td>FF</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB_X19_Y15_N4</td>
</tr>
<tr>
<td>7.929</td>
<td>0.000</td>
<td>0.178</td>
<td>FF</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB_X19_Y15_N4</td>
</tr>
<tr>
<td>7.929</td>
<td>0.000</td>
<td>0.178</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB_X19_Y15_N6</td>
</tr>
<tr>
<td>9.089</td>
<td>0.000</td>
<td>0.178</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB_X19_Y15_N8</td>
</tr>
<tr>
<td>8.089</td>
<td>0.000</td>
<td>0.178</td>
<td>FR</td>
<td>CELL</td>
<td>1</td>
<td>LCCOMB_X19_Y15_N8</td>
</tr>
<tr>
<td>8.089</td>
<td>0.080</td>
<td>0.178</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB_X19_Y15_N10</td>
</tr>
<tr>
<td>8.169</td>
<td>0.080</td>
<td>0.178</td>
<td>FR</td>
<td>CELL</td>
<td>1</td>
<td>LCCOMB_X19_Y15_N10</td>
</tr>
<tr>
<td>8.169</td>
<td>0.000</td>
<td>0.178</td>
<td>FF</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB_X19_Y15_N12</td>
</tr>
<tr>
<td>8.249</td>
<td>0.080</td>
<td>0.178</td>
<td>FR</td>
<td>CELL</td>
<td>1</td>
<td>LCCOMB_X19_Y15_N12</td>
</tr>
<tr>
<td>8.249</td>
<td>0.000</td>
<td>0.178</td>
<td>FR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB_X19_Y15_N14</td>
</tr>
<tr>
<td>8.423</td>
<td>0.174</td>
<td>0.178</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB_X19_Y15_N14</td>
</tr>
<tr>
<td>8.423</td>
<td>0.000</td>
<td>0.178</td>
<td>FR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB_X19_Y15_N16</td>
</tr>
<tr>
<td>8.503</td>
<td>0.080</td>
<td>0.178</td>
<td>FR</td>
<td>CELL</td>
<td>1</td>
<td>LCCOMB_X19_Y15_N16</td>
</tr>
<tr>
<td>8.503</td>
<td>0.080</td>
<td>0.178</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB_X19_Y15_N18</td>
</tr>
<tr>
<td>8.583</td>
<td>0.080</td>
<td>0.178</td>
<td>RF</td>
<td>CELL</td>
<td>1</td>
<td>LCCOMB_X19_Y15_N18</td>
</tr>
<tr>
<td>8.583</td>
<td>0.080</td>
<td>0.178</td>
<td>FR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB_X19_Y15_N20</td>
</tr>
<tr>
<td>8.663</td>
<td>0.080</td>
<td>0.178</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB_X19_Y15_N20</td>
</tr>
<tr>
<td>8.663</td>
<td>0.000</td>
<td>0.178</td>
<td>FR</td>
<td>CELL</td>
<td>1</td>
<td>LCCOMB_X19_Y15_N22</td>
</tr>
<tr>
<td>8.743</td>
<td>0.080</td>
<td>0.178</td>
<td>FR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB_X19_Y15_N22</td>
</tr>
<tr>
<td>8.743</td>
<td>0.080</td>
<td>0.178</td>
<td>RF</td>
<td>CELL</td>
<td>1</td>
<td>LCCOMB_X19_Y15_N24</td>
</tr>
<tr>
<td>8.823</td>
<td>0.080</td>
<td>0.178</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB_X19_Y15_N24</td>
</tr>
<tr>
<td>8.823</td>
<td>0.000</td>
<td>0.178</td>
<td>FR</td>
<td>CELL</td>
<td>1</td>
<td>LCCOMB_X19_Y15_N26</td>
</tr>
<tr>
<td>8.903</td>
<td>0.080</td>
<td>0.178</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB_X19_Y15_N26</td>
</tr>
<tr>
<td>Total</td>
<td>Incr</td>
<td>RF</td>
<td>Type</td>
<td>Fanout</td>
<td>Location</td>
<td>Element</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>----</td>
<td>------</td>
<td>--------</td>
<td>----------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>18.667</td>
<td>15.810</td>
<td></td>
<td>uTco</td>
<td>1</td>
<td>LCFF X21_Y16_N25</td>
<td>inst_rom:rom</td>
</tr>
<tr>
<td>3.134</td>
<td>0.277</td>
<td>FF</td>
<td>CELL</td>
<td>23</td>
<td>LCFF X21_Y16_N25</td>
<td>rom</td>
</tr>
<tr>
<td>3.754</td>
<td>0.620</td>
<td>FF</td>
<td>IC</td>
<td>1</td>
<td>LCCOMB X20_Y16_N20</td>
<td>ctrl</td>
</tr>
<tr>
<td>3.931</td>
<td>0.177</td>
<td>FR</td>
<td>CELL</td>
<td>29</td>
<td>LCCOMB X20_Y16_N20</td>
<td>ctrl</td>
</tr>
<tr>
<td>4.877</td>
<td>0.946</td>
<td>RR</td>
<td>IC</td>
<td>1</td>
<td>LCCOMB X19_Y12_N4</td>
<td>muxImmediateMode</td>
</tr>
<tr>
<td>5.055</td>
<td>0.178</td>
<td>RR</td>
<td>CELL</td>
<td>5</td>
<td>LCCOMB X19_Y12_N4</td>
<td>muxImmediateMode</td>
</tr>
<tr>
<td>5.643</td>
<td>0.588</td>
<td>RR</td>
<td>IC</td>
<td>1</td>
<td>LCCOMB X18_Y12_N20</td>
<td>theAlu</td>
</tr>
<tr>
<td>6.188</td>
<td>0.545</td>
<td>RR</td>
<td>CELL</td>
<td>1</td>
<td>LCCOMB X18_Y12_N20</td>
<td>theAlu</td>
</tr>
<tr>
<td>7.094</td>
<td>0.906</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB X19_Y16_N30</td>
<td>theAlu</td>
</tr>
<tr>
<td>7.689</td>
<td>0.595</td>
<td>R</td>
<td>IC</td>
<td>1</td>
<td>LCCOMB X19_Y16_N30</td>
<td>theAlu</td>
</tr>
<tr>
<td>7.689</td>
<td>0.000</td>
<td>FF</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB X19_Y15_N0</td>
<td>theAlu</td>
</tr>
<tr>
<td>7.679</td>
<td>0.080</td>
<td>FR</td>
<td>CELL</td>
<td>1</td>
<td>LCCOMB X19_Y15_N0</td>
<td>theAlu</td>
</tr>
<tr>
<td>7.679</td>
<td>0.080</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB X19_Y15_N4</td>
<td>theAlu</td>
</tr>
<tr>
<td>7.929</td>
<td>0.080</td>
<td>RR</td>
<td>IC</td>
<td>1</td>
<td>LCCOMB X19_Y15_N4</td>
<td>theAlu</td>
</tr>
<tr>
<td>7.929</td>
<td>0.000</td>
<td>FF</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB X19_Y15_N6</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.009</td>
<td>0.080</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB X19_Y15_N6</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.009</td>
<td>0.080</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB X19_Y15_N8</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.089</td>
<td>0.080</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB X19_Y15_N8</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.169</td>
<td>0.080</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB X19_Y15_N10</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.169</td>
<td>0.080</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB X19_Y15_N10</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.249</td>
<td>0.080</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB X19_Y15_N16</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.249</td>
<td>0.080</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB X19_Y15_N16</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.429</td>
<td>0.174</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB X19_Y15_N14</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.429</td>
<td>0.174</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB X19_Y15_N14</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.423</td>
<td>0.174</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB X19_Y15_N16</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.423</td>
<td>0.174</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB X19_Y15_N16</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.503</td>
<td>0.080</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB X19_Y15_N18</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.503</td>
<td>0.080</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB X19_Y15_N18</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.583</td>
<td>0.080</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB X19_Y15_N22</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.583</td>
<td>0.080</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB X19_Y15_N22</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.663</td>
<td>0.000</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB X19_Y15_N20</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.663</td>
<td>0.000</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB X19_Y15_N20</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.823</td>
<td>0.080</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB X19_Y15_N24</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.823</td>
<td>0.080</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB X19_Y15_N24</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.743</td>
<td>0.080</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB X19_Y15_N22</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.743</td>
<td>0.080</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB X19_Y15_N22</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.903</td>
<td>0.080</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB X19_Y15_N26</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.903</td>
<td>0.080</td>
<td>RR</td>
<td>IC</td>
<td>2</td>
<td>LCCOMB X19_Y15_N26</td>
<td>theAlu</td>
</tr>
</tbody>
</table>

Imem 2.77 ns
<table>
<thead>
<tr>
<th>Total</th>
<th>Incr</th>
<th>RF</th>
<th>Type</th>
<th>Fanout</th>
<th>Location</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.667</td>
<td>15.810</td>
<td>0.277</td>
<td>uTco</td>
<td>1</td>
<td>LCFF</td>
<td>X21_Y16_N25</td>
</tr>
<tr>
<td>3.134</td>
<td>0.000</td>
<td>0.000</td>
<td>FF</td>
<td>23</td>
<td>LCFF</td>
<td>X21_Y16_N25</td>
</tr>
<tr>
<td>3.754</td>
<td>0.620</td>
<td>1</td>
<td>IC</td>
<td>1</td>
<td>LCCMB</td>
<td>X20_Y16_N20</td>
</tr>
<tr>
<td>3.931</td>
<td>0.177</td>
<td>0.000</td>
<td>FF</td>
<td>29</td>
<td>LCCMB</td>
<td>X20_Y16_N20</td>
</tr>
<tr>
<td>4.877</td>
<td>0.946</td>
<td>0.000</td>
<td>RR</td>
<td>1</td>
<td>LCCOMB</td>
<td>X19_Y12_N4</td>
</tr>
<tr>
<td>5.055</td>
<td>0.178</td>
<td>0.000</td>
<td>RR</td>
<td>5</td>
<td>LCCOMB</td>
<td>X19_Y12_N4</td>
</tr>
<tr>
<td>5.643</td>
<td>0.588</td>
<td>0.000</td>
<td>FF</td>
<td>1</td>
<td>LCCOMB</td>
<td>X18_Y12_N20</td>
</tr>
<tr>
<td>6.188</td>
<td>0.545</td>
<td>0.000</td>
<td>RR</td>
<td>1</td>
<td>LCCOMB</td>
<td>X18_Y12_N20</td>
</tr>
<tr>
<td>7.094</td>
<td>0.906</td>
<td>0.000</td>
<td>RR</td>
<td>2</td>
<td>LCCOMB</td>
<td>X19_Y12_N30</td>
</tr>
<tr>
<td>7.689</td>
<td>0.595</td>
<td>0.000</td>
<td>RR</td>
<td>1</td>
<td>LCCOMB</td>
<td>X19_Y12_N30</td>
</tr>
<tr>
<td>7.689</td>
<td>0.000</td>
<td>0.000</td>
<td>FF</td>
<td>2</td>
<td>LCCOMB</td>
<td>X19_Y15_N0</td>
</tr>
<tr>
<td>7.769</td>
<td>0.080</td>
<td>0.000</td>
<td>FR</td>
<td>1</td>
<td>LCCOMB</td>
<td>X19_Y15_N0</td>
</tr>
<tr>
<td>7.769</td>
<td>0.080</td>
<td>0.000</td>
<td>FR</td>
<td>2</td>
<td>LCCOMB</td>
<td>X19_Y15_N4</td>
</tr>
<tr>
<td>7.929</td>
<td>0.080</td>
<td>0.000</td>
<td>RR</td>
<td>1</td>
<td>LCCOMB</td>
<td>X19_Y15_N4</td>
</tr>
<tr>
<td>7.929</td>
<td>0.080</td>
<td>0.000</td>
<td>RR</td>
<td>2</td>
<td>LCCOMB</td>
<td>X19_Y15_N6</td>
</tr>
<tr>
<td>8.009</td>
<td>0.080</td>
<td>0.000</td>
<td>RR</td>
<td>1</td>
<td>LCCOMB</td>
<td>X19_Y15_N6</td>
</tr>
<tr>
<td>8.009</td>
<td>0.080</td>
<td>0.000</td>
<td>RR</td>
<td>2</td>
<td>LCCOMB</td>
<td>X19_Y15_N6</td>
</tr>
<tr>
<td>8.089</td>
<td>0.080</td>
<td>0.000</td>
<td>RF</td>
<td>1</td>
<td>LCCOMB</td>
<td>X19_Y15_N8</td>
</tr>
<tr>
<td>8.089</td>
<td>0.080</td>
<td>0.000</td>
<td>RF</td>
<td>2</td>
<td>LCCOMB</td>
<td>X19_Y15_N8</td>
</tr>
<tr>
<td>8.169</td>
<td>0.080</td>
<td>0.000</td>
<td>RF</td>
<td>1</td>
<td>LCCOMB</td>
<td>X19_Y15_N10</td>
</tr>
<tr>
<td>8.169</td>
<td>0.080</td>
<td>0.000</td>
<td>RF</td>
<td>2</td>
<td>LCCOMB</td>
<td>X19_Y15_N10</td>
</tr>
<tr>
<td>8.249</td>
<td>0.080</td>
<td>0.000</td>
<td>FR</td>
<td>1</td>
<td>LCCOMB</td>
<td>X19_Y15_N12</td>
</tr>
<tr>
<td>8.249</td>
<td>0.080</td>
<td>0.000</td>
<td>FR</td>
<td>2</td>
<td>LCCOMB</td>
<td>X19_Y15_N12</td>
</tr>
<tr>
<td>8.423</td>
<td>0.174</td>
<td>0.000</td>
<td>RR</td>
<td>1</td>
<td>LCCOMB</td>
<td>X19_Y15_N14</td>
</tr>
<tr>
<td>8.423</td>
<td>0.174</td>
<td>0.000</td>
<td>RR</td>
<td>2</td>
<td>LCCOMB</td>
<td>X19_Y15_N14</td>
</tr>
<tr>
<td>8.423</td>
<td>0.174</td>
<td>0.000</td>
<td>RR</td>
<td>2</td>
<td>LCCOMB</td>
<td>X19_Y15_N16</td>
</tr>
<tr>
<td>8.503</td>
<td>0.080</td>
<td>0.000</td>
<td>RF</td>
<td>1</td>
<td>LCCOMB</td>
<td>X19_Y15_N16</td>
</tr>
<tr>
<td>8.503</td>
<td>0.080</td>
<td>0.000</td>
<td>RF</td>
<td>2</td>
<td>LCCOMB</td>
<td>X19_Y15_N16</td>
</tr>
<tr>
<td>8.583</td>
<td>0.080</td>
<td>0.000</td>
<td>RF</td>
<td>1</td>
<td>LCCOMB</td>
<td>X19_Y15_N18</td>
</tr>
<tr>
<td>8.583</td>
<td>0.080</td>
<td>0.000</td>
<td>RF</td>
<td>2</td>
<td>LCCOMB</td>
<td>X19_Y15_N18</td>
</tr>
<tr>
<td>8.663</td>
<td>0.080</td>
<td>0.000</td>
<td>RR</td>
<td>1</td>
<td>LCCOMB</td>
<td>X19_Y15_N20</td>
</tr>
<tr>
<td>8.663</td>
<td>0.080</td>
<td>0.000</td>
<td>RR</td>
<td>2</td>
<td>LCCOMB</td>
<td>X19_Y15_N20</td>
</tr>
<tr>
<td>8.743</td>
<td>0.080</td>
<td>0.000</td>
<td>RF</td>
<td>1</td>
<td>LCCOMB</td>
<td>X19_Y15_N22</td>
</tr>
<tr>
<td>8.743</td>
<td>0.080</td>
<td>0.000</td>
<td>RF</td>
<td>2</td>
<td>LCCOMB</td>
<td>X19_Y15_N22</td>
</tr>
<tr>
<td>8.823</td>
<td>0.080</td>
<td>0.000</td>
<td>RR</td>
<td>1</td>
<td>LCCOMB</td>
<td>X19_Y15_N24</td>
</tr>
<tr>
<td>8.823</td>
<td>0.080</td>
<td>0.000</td>
<td>RR</td>
<td>2</td>
<td>LCCOMB</td>
<td>X19_Y15_N24</td>
</tr>
<tr>
<td>8.823</td>
<td>0.080</td>
<td>0.000</td>
<td>RR</td>
<td>2</td>
<td>LCCOMB</td>
<td>X19_Y15_N26</td>
</tr>
<tr>
<td>8.823</td>
<td>0.080</td>
<td>0.000</td>
<td>RR</td>
<td>2</td>
<td>LCCOMB</td>
<td>X19_Y15_N26</td>
</tr>
<tr>
<td>8.903</td>
<td>0.080</td>
<td>0.000</td>
<td>RF</td>
<td>1</td>
<td>LCCOMB</td>
<td>X19_Y15_N28</td>
</tr>
<tr>
<td>8.903</td>
<td>0.080</td>
<td>0.000</td>
<td>RF</td>
<td>2</td>
<td>LCCOMB</td>
<td>X19_Y15_N28</td>
</tr>
</tbody>
</table>

Imem 2.77 ns
Ctrl 0.797 ns
<table>
<thead>
<tr>
<th>Total</th>
<th>Incr</th>
<th>RF</th>
<th>Type</th>
<th>Fanout</th>
<th>Location</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.667</td>
<td>15.810</td>
<td>0.277</td>
<td>uTco</td>
<td>1</td>
<td>X21_Y16_N25</td>
<td>inst_rom:rom</td>
</tr>
<tr>
<td>3.134</td>
<td>0.000</td>
<td>0.277</td>
<td>FF</td>
<td>23</td>
<td>X21_Y16_N25</td>
<td>rom</td>
</tr>
<tr>
<td>3.754</td>
<td>0.620</td>
<td>0.177</td>
<td>FF</td>
<td>1</td>
<td>X20_Y16_N20</td>
<td>ctrl</td>
</tr>
<tr>
<td>3.931</td>
<td>0.177</td>
<td>0.080</td>
<td>IC</td>
<td>29</td>
<td>X20_Y16_N20</td>
<td>ctrl</td>
</tr>
<tr>
<td>4.877</td>
<td>0.946</td>
<td>0.080</td>
<td>IC</td>
<td>1</td>
<td>X19_Y12_N4</td>
<td>muxImmed</td>
</tr>
<tr>
<td>5.055</td>
<td>0.178</td>
<td>0.080</td>
<td>RR</td>
<td>5</td>
<td>X19_Y12_N4</td>
<td>muxImmediate</td>
</tr>
<tr>
<td>5.643</td>
<td>0.588</td>
<td>0.178</td>
<td>RR</td>
<td>1</td>
<td>X18_Y12_N20</td>
<td>theAlu</td>
</tr>
<tr>
<td>6.188</td>
<td>0.545</td>
<td>0.080</td>
<td>RR</td>
<td>1</td>
<td>X18_Y12_N20</td>
<td>theAlu</td>
</tr>
<tr>
<td>7.094</td>
<td>0.906</td>
<td>0.080</td>
<td>RR</td>
<td>1</td>
<td>X19_Y15_N0</td>
<td>theAlu</td>
</tr>
<tr>
<td>7.689</td>
<td>0.595</td>
<td>0.080</td>
<td>RR</td>
<td>1</td>
<td>X19_Y15_N0</td>
<td>theAlu</td>
</tr>
<tr>
<td>7.929</td>
<td>0.080</td>
<td>0.080</td>
<td>IC</td>
<td>2</td>
<td>X19_Y15_N4</td>
<td>theAlu</td>
</tr>
<tr>
<td>7.929</td>
<td>0.080</td>
<td>0.080</td>
<td>IC</td>
<td>2</td>
<td>X19_Y15_N4</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.009</td>
<td>0.080</td>
<td>0.080</td>
<td>IC</td>
<td>2</td>
<td>X19_Y15_N4</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.009</td>
<td>0.080</td>
<td>0.080</td>
<td>IC</td>
<td>2</td>
<td>X19_Y15_N4</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.069</td>
<td>0.080</td>
<td>0.080</td>
<td>IC</td>
<td>2</td>
<td>X19_Y15_N6</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.069</td>
<td>0.080</td>
<td>0.080</td>
<td>IC</td>
<td>2</td>
<td>X19_Y15_N6</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.169</td>
<td>0.080</td>
<td>0.080</td>
<td>IC</td>
<td>2</td>
<td>X19_Y15_N6</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.169</td>
<td>0.080</td>
<td>0.080</td>
<td>IC</td>
<td>2</td>
<td>X19_Y15_N6</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.249</td>
<td>0.080</td>
<td>0.080</td>
<td>IC</td>
<td>2</td>
<td>X19_Y15_N6</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.249</td>
<td>0.080</td>
<td>0.080</td>
<td>IC</td>
<td>2</td>
<td>X19_Y15_N6</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.423</td>
<td>0.174</td>
<td>0.080</td>
<td>IC</td>
<td>2</td>
<td>X18_Y15_N6</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.423</td>
<td>0.174</td>
<td>0.080</td>
<td>IC</td>
<td>2</td>
<td>X18_Y15_N6</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.583</td>
<td>0.080</td>
<td>0.080</td>
<td>IC</td>
<td>2</td>
<td>X19_Y15_N12</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.583</td>
<td>0.080</td>
<td>0.080</td>
<td>IC</td>
<td>2</td>
<td>X19_Y15_N12</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.663</td>
<td>0.080</td>
<td>0.080</td>
<td>IC</td>
<td>2</td>
<td>X19_Y15_N22</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.663</td>
<td>0.080</td>
<td>0.080</td>
<td>IC</td>
<td>2</td>
<td>X19_Y15_N22</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.823</td>
<td>0.080</td>
<td>0.080</td>
<td>IC</td>
<td>2</td>
<td>X19_Y15_N22</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.823</td>
<td>0.080</td>
<td>0.080</td>
<td>IC</td>
<td>2</td>
<td>X19_Y15_N22</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.908</td>
<td>0.080</td>
<td>0.080</td>
<td>IC</td>
<td>2</td>
<td>X19_Y15_N22</td>
<td>theAlu</td>
</tr>
<tr>
<td>8.908</td>
<td>0.080</td>
<td>0.080</td>
<td>IC</td>
<td>2</td>
<td>X19_Y15_N22</td>
<td>theAlu</td>
</tr>
</tbody>
</table>

- **Ctrl**: 0.797 ns
- **Imem**: 2.77 ns
- **ArgBMux**: 1.124 ns
<table>
<thead>
<tr>
<th>Total</th>
<th>Incr</th>
<th>RF</th>
<th>Type</th>
<th>Fanout</th>
<th>Location</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.667</td>
<td>15.810</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>data path</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Ctrl 0.797 ns</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Imem 2.77 ns</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ArgBMux 1.124 ns</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ALU 6.527 ns</td>
</tr>
<tr>
<td>Time</td>
<td>Type</td>
<td>Cell</td>
<td>Location</td>
<td>Function</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------</td>
<td>------------------------------</td>
<td>---------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.983</td>
<td>RF</td>
<td>CELL</td>
<td>LCCOMB_X19_Y15_N28</td>
<td>theAlu</td>
<td>Add0~44</td>
<td>cout</td>
</tr>
<tr>
<td>9.144</td>
<td>RF</td>
<td>CELL</td>
<td>LCCOMB_X19_Y15_N30</td>
<td>theAlu</td>
<td>Add0~46</td>
<td>cin</td>
</tr>
<tr>
<td>9.144</td>
<td>FF</td>
<td>IC</td>
<td>LCCOMB_X19_Y14_N0</td>
<td>theAlu</td>
<td>Add0~48</td>
<td>cin</td>
</tr>
<tr>
<td>9.224</td>
<td>RF</td>
<td>CELL</td>
<td>LCCOMB_X19_Y14_N0</td>
<td>theAlu</td>
<td>Add0~50</td>
<td>cin</td>
</tr>
<tr>
<td>9.224</td>
<td>FF</td>
<td>IC</td>
<td>LCCOMB_X19_Y14_N0</td>
<td>theAlu</td>
<td>Add0~52</td>
<td>cin</td>
</tr>
<tr>
<td>9.384</td>
<td>RF</td>
<td>CELL</td>
<td>LCCOMB_X19_Y14_N2</td>
<td>theAlu</td>
<td>Add0~54</td>
<td>cin</td>
</tr>
<tr>
<td>9.384</td>
<td>FF</td>
<td>IC</td>
<td>LCCOMB_X19_Y14_N2</td>
<td>theAlu</td>
<td>Add0~56</td>
<td>cin</td>
</tr>
<tr>
<td>9.464</td>
<td>RF</td>
<td>CELL</td>
<td>LCCOMB_X19_Y14_N6</td>
<td>theAlu</td>
<td>Add0~58</td>
<td>cin</td>
</tr>
<tr>
<td>9.464</td>
<td>FF</td>
<td>IC</td>
<td>LCCOMB_X19_Y14_N6</td>
<td>theAlu</td>
<td>Add0~60</td>
<td>cin</td>
</tr>
<tr>
<td>9.544</td>
<td>RF</td>
<td>CELL</td>
<td>LCCOMB_X19_Y14_N8</td>
<td>theAlu</td>
<td>Add0~62</td>
<td>cin</td>
</tr>
<tr>
<td>9.544</td>
<td>FF</td>
<td>IC</td>
<td>LCCOMB_X19_Y14_N8</td>
<td>theAlu</td>
<td>Add0~64</td>
<td>cin</td>
</tr>
<tr>
<td>10.336</td>
<td>RF</td>
<td>CELL</td>
<td>LCCOMB_X19_Y14_N10</td>
<td>theAlu</td>
<td>Add0~66</td>
<td>cin</td>
</tr>
<tr>
<td>11.260</td>
<td>FF</td>
<td>IC</td>
<td>LCCOMB_X19_Y14_N10</td>
<td>theAlu</td>
<td>Add0~68</td>
<td>cin</td>
</tr>
<tr>
<td>11.582</td>
<td>RF</td>
<td>CELL</td>
<td>LCCOMB_X19_Y14_N12</td>
<td>theAlu</td>
<td>Add0~70</td>
<td>cin</td>
</tr>
<tr>
<td>11.582</td>
<td>FF</td>
<td>IC</td>
<td>LCCOMB_X19_Y14_N12</td>
<td>theAlu</td>
<td>Add0~72</td>
<td>cin</td>
</tr>
<tr>
<td>12.517</td>
<td>FF</td>
<td>IC</td>
<td>LCCOMB_X20_Y15_N4</td>
<td>dataMem</td>
<td>Equal0~0</td>
<td>datad</td>
</tr>
<tr>
<td>12.694</td>
<td>FF</td>
<td>IC</td>
<td>LCCOMB_X20_Y15_N4</td>
<td>dataMem</td>
<td>Equal0~0</td>
<td>combout</td>
</tr>
<tr>
<td>13.004</td>
<td>FF</td>
<td>IC</td>
<td>LCCOMB_X20_Y15_N4</td>
<td>dataMem</td>
<td>Equal0~0</td>
<td>datad</td>
</tr>
<tr>
<td>13.326</td>
<td>FF</td>
<td>IC</td>
<td>LCCOMB_X20_Y15_N4</td>
<td>dataMem</td>
<td>Equal0~0</td>
<td>combout</td>
</tr>
<tr>
<td>14.615</td>
<td>RR</td>
<td>IC</td>
<td>LCCOMB_X18_Y13_N4</td>
<td>muxWriteRegData</td>
<td>O_out[3]~3</td>
<td>datad</td>
</tr>
<tr>
<td>14.793</td>
<td>RR</td>
<td>IC</td>
<td>LCCOMB_X18_Y13_N4</td>
<td>muxWriteRegData</td>
<td>O_out[3]~3</td>
<td>combout</td>
</tr>
<tr>
<td>16.071</td>
<td>RR</td>
<td>IC</td>
<td>LCCOMB_X19_Y18_N24</td>
<td>muxWriteRegData</td>
<td>O_out[2]</td>
<td>datad</td>
</tr>
<tr>
<td>16.393</td>
<td>RR</td>
<td>IC</td>
<td>LCCOMB_X19_Y18_N24</td>
<td>muxWriteRegData</td>
<td>O_out[2]</td>
<td>combout</td>
</tr>
<tr>
<td>17.243</td>
<td>RR</td>
<td>IC</td>
<td>LCCOMB_X18_Y16_N0</td>
<td>rf</td>
<td>regs~8</td>
<td>datad</td>
</tr>
<tr>
<td>17.421</td>
<td>RR</td>
<td>CELL</td>
<td>LCCOMB_X18_Y16_N0</td>
<td>rf</td>
<td>regs~8</td>
<td>combout</td>
</tr>
<tr>
<td>18.254</td>
<td>RR</td>
<td>IC</td>
<td>LCFF_X20_Y16_N27</td>
<td>rf</td>
<td>regs[0][2]</td>
<td>sdata</td>
</tr>
<tr>
<td>18.667</td>
<td>RR</td>
<td>CELL</td>
<td>LCFF_X20_Y16_N27</td>
<td>register_file:rf</td>
<td>regs[0][2]</td>
<td></td>
</tr>
</tbody>
</table>

Dmem  1.744 ns
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Type</th>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8983</td>
<td>0x080</td>
<td>FR</td>
<td>CELL 1</td>
<td>LCCOMB_X19_Y15_N28 theAlu</td>
</tr>
<tr>
<td>0x983</td>
<td>0x000</td>
<td>RR</td>
<td>IC 2</td>
<td>LCCOMB_X19_Y15_N30 theAlu</td>
</tr>
<tr>
<td>0x9144</td>
<td>0x161</td>
<td>RF</td>
<td>CELL 1</td>
<td>LCCOMB_X19_Y15_N30 theAlu</td>
</tr>
<tr>
<td>0x9144</td>
<td>0x000</td>
<td>FF</td>
<td>IC 2</td>
<td>LCCOMB_X19_Y14_N0 theAlu</td>
</tr>
<tr>
<td>0x9224</td>
<td>0x080</td>
<td>FR</td>
<td>CELL 1</td>
<td>LCCOMB_X19_Y14_N0 theAlu</td>
</tr>
<tr>
<td>0x9224</td>
<td>0x000</td>
<td>RR</td>
<td>IC 2</td>
<td>LCCOMB_X19_Y14_N0 theAlu</td>
</tr>
<tr>
<td>0x9304</td>
<td>0x080</td>
<td>RF</td>
<td>CELL 1</td>
<td>LCCOMB_X19_Y14_N2 theAlu</td>
</tr>
<tr>
<td>0x9304</td>
<td>0x000</td>
<td>FF</td>
<td>IC 2</td>
<td>LCCOMB_X19_Y14_N4 theAlu</td>
</tr>
<tr>
<td>0x9384</td>
<td>0x080</td>
<td>FR</td>
<td>CELL 1</td>
<td>LCCOMB_X19_Y14_N4 theAlu</td>
</tr>
<tr>
<td>0x9384</td>
<td>0x000</td>
<td>RR</td>
<td>IC 2</td>
<td>LCCOMB_X19_Y14_N4 theAlu</td>
</tr>
<tr>
<td>0x9464</td>
<td>0x080</td>
<td>RF</td>
<td>CELL 1</td>
<td>LCCOMB_X19_Y14_N6 theAlu</td>
</tr>
<tr>
<td>0x9464</td>
<td>0x000</td>
<td>FF</td>
<td>IC 2</td>
<td>LCCOMB_X19_Y14_N8 theAlu</td>
</tr>
<tr>
<td>0x9544</td>
<td>0x080</td>
<td>FR</td>
<td>CELL 1</td>
<td>LCCOMB_X19_Y14_N8 theAlu</td>
</tr>
<tr>
<td>0x9544</td>
<td>0x000</td>
<td>RR</td>
<td>IC 2</td>
<td>LCCOMB_X19_Y14_N8 theAlu</td>
</tr>
<tr>
<td>0x9624</td>
<td>0x080</td>
<td>RF</td>
<td>CELL 1</td>
<td>LCCOMB_X19_Y14_N10 theAlu</td>
</tr>
<tr>
<td>0x9624</td>
<td>0x000</td>
<td>FF</td>
<td>IC 2</td>
<td>LCCOMB_X19_Y14_N10 theAlu</td>
</tr>
<tr>
<td>0x9624</td>
<td>0x000</td>
<td>RR</td>
<td>IC 2</td>
<td>LCCOMB_X19_Y14_N10 theAlu</td>
</tr>
<tr>
<td>0x9624</td>
<td>0x000</td>
<td>FF</td>
<td>IC 2</td>
<td>LCCOMB_X19_Y14_N10 theAlu</td>
</tr>
<tr>
<td>0x9704</td>
<td>0x080</td>
<td>FR</td>
<td>CELL 1</td>
<td>LCCOMB_X19_Y14_N12 theAlu</td>
</tr>
<tr>
<td>0x9704</td>
<td>0x000</td>
<td>RR</td>
<td>IC 2</td>
<td>LCCOMB_X19_Y14_N12 theAlu</td>
</tr>
<tr>
<td>0x9878</td>
<td>0x174</td>
<td>RF</td>
<td>CELL 1</td>
<td>LCCOMB_X19_Y14_N12 theAlu</td>
</tr>
<tr>
<td>0x9878</td>
<td>0x000</td>
<td>FF</td>
<td>IC 2</td>
<td>LCCOMB_X19_Y14_N12 theAlu</td>
</tr>
<tr>
<td>0x10336</td>
<td>0x458</td>
<td>FF</td>
<td>CELL 1</td>
<td>LCCOMB_X19_Y14_N16 theAlu</td>
</tr>
<tr>
<td>0x11260</td>
<td>0x924</td>
<td>FF</td>
<td>IC 1</td>
<td>LCCOMB_X19_Y18_N0 theAlu</td>
</tr>
<tr>
<td>0x11582</td>
<td>0x322</td>
<td>FF</td>
<td>CELL 12</td>
<td>LCCOMB_X19_Y18_N0 theAlu</td>
</tr>
<tr>
<td>0x12517</td>
<td>0x935</td>
<td>FF</td>
<td>IC 1</td>
<td>LCCOMB_X20_Y15_N4 dataMem</td>
</tr>
<tr>
<td>0x12694</td>
<td>0x177</td>
<td>FR</td>
<td>CELL 1</td>
<td>LCCOMB_X20_Y15_N4 dataMem</td>
</tr>
<tr>
<td>0x13004</td>
<td>0x310</td>
<td>RR</td>
<td>IC 1</td>
<td>LCCOMB_X20_Y15_N4 dataMem</td>
</tr>
<tr>
<td>0x13326</td>
<td>0x322</td>
<td>RR</td>
<td>CELL 31</td>
<td>LCCOMB_X20_Y15_N12 dataMem</td>
</tr>
<tr>
<td>0x14615</td>
<td>1.289</td>
<td>RR</td>
<td>IC 1</td>
<td>LCCOMB_X18_Y13_N4 muxWriteRegData</td>
</tr>
<tr>
<td>0x14793</td>
<td>0.178</td>
<td>RR</td>
<td>CELL 7</td>
<td>LCCOMB_X18_Y13_N4 muxWriteRegData</td>
</tr>
<tr>
<td>0x16071</td>
<td>1.278</td>
<td>RR</td>
<td>IC 1</td>
<td>LCCOMB_X19_Y18_N24 muxWriteRegData</td>
</tr>
<tr>
<td>0x16393</td>
<td>0.322</td>
<td>RR</td>
<td>CELL 1</td>
<td>LCCOMB_X19_Y18_N24 muxWriteRegData</td>
</tr>
<tr>
<td>0x17243</td>
<td>0.850</td>
<td>RR</td>
<td>IC 1</td>
<td>LCCOMB_X18_Y18_N0 rf</td>
</tr>
<tr>
<td>0x17421</td>
<td>0.178</td>
<td>RR</td>
<td>CELL 3</td>
<td>LCCOMB_X18_Y16_N0 rf</td>
</tr>
<tr>
<td>0x18254</td>
<td>0.833</td>
<td>RR</td>
<td>IC 1</td>
<td>LCFF_X20_Y16_N27 rf</td>
</tr>
<tr>
<td>0x18667</td>
<td>0.413</td>
<td>RR</td>
<td>CELL 1</td>
<td>LCFF_X20_Y16_N27 register_file:rf</td>
</tr>
</tbody>
</table>


<p>| ALU 6.527 ns |</p>
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Type</th>
<th>Input/Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.983</td>
<td>0.080</td>
<td>FR</td>
<td>CELL 1</td>
<td>LCCOMB_X19_Y15_N28 theAlu[Add0~44]cout</td>
</tr>
<tr>
<td>9.144</td>
<td>0.000</td>
<td>RR</td>
<td>IC 2</td>
<td>LCCOMB_X19_Y15_N30 theAlu[Add0~46]cin</td>
</tr>
<tr>
<td>9.144</td>
<td>0.000</td>
<td>RF</td>
<td>CELL 1</td>
<td>LCCOMB_X19_Y15_N30 theAlu[Add0~46]cout</td>
</tr>
<tr>
<td>9.224</td>
<td>0.000</td>
<td>FF</td>
<td>IC 2</td>
<td>LCCOMB_X19_Y14_N0 theAlu[Add0~48]cin</td>
</tr>
<tr>
<td>9.224</td>
<td>0.000</td>
<td>RR</td>
<td>IC 2</td>
<td>LCCOMB_X19_Y14_N0 theAlu[Add0~48]cout</td>
</tr>
<tr>
<td>9.304</td>
<td>0.080</td>
<td>RR</td>
<td>IC 1</td>
<td>LCCOMB_X19_Y14_N2 theAlu[Add0~50]cin</td>
</tr>
<tr>
<td>9.304</td>
<td>0.080</td>
<td>RF</td>
<td>IC 2</td>
<td>LCCOMB_X19_Y14_N2 theAlu[Add0~50]cout</td>
</tr>
<tr>
<td>9.384</td>
<td>0.000</td>
<td>RR</td>
<td>IC 2</td>
<td>LCCOMB_X19_Y14_N4 theAlu[Add0~52]cout</td>
</tr>
<tr>
<td>9.464</td>
<td>0.080</td>
<td>RF</td>
<td>IC 1</td>
<td>LCCOMB_X19_Y14_N6 theAlu[Add0~54]cout</td>
</tr>
<tr>
<td>9.464</td>
<td>0.080</td>
<td>FF</td>
<td>IC 2</td>
<td>LCCOMB_X19_Y14_N6 theAlu[Add0~54]cout</td>
</tr>
<tr>
<td>9.544</td>
<td>0.000</td>
<td>RR</td>
<td>IC 2</td>
<td>LCCOMB_X19_Y14_N8 theAlu[Add0~56]cout</td>
</tr>
<tr>
<td>9.544</td>
<td>0.080</td>
<td>FR</td>
<td>IC 2</td>
<td>LCCOMB_X19_Y14_N8 theAlu[Add0~56]cin</td>
</tr>
<tr>
<td>9.624</td>
<td>0.000</td>
<td>RR</td>
<td>IC 2</td>
<td>LCCOMB_X19_Y14_N10 theAlu[Add0~58]cout</td>
</tr>
<tr>
<td>9.624</td>
<td>0.000</td>
<td>FF</td>
<td>IC 2</td>
<td>LCCOMB_X19_Y14_N10 theAlu[Add0~58]cout</td>
</tr>
<tr>
<td>9.704</td>
<td>0.080</td>
<td>RR</td>
<td>IC 1</td>
<td>LCCOMB_X19_Y14_N12 theAlu[Add0~60]cout</td>
</tr>
<tr>
<td>9.704</td>
<td>0.080</td>
<td>RF</td>
<td>IC 2</td>
<td>LCCOMB_X19_Y14_N12 theAlu[Add0~60]cout</td>
</tr>
<tr>
<td>9.878</td>
<td>0.174</td>
<td>RF</td>
<td>CELL 1</td>
<td>LCCOMB_X19_Y14_N14 theAlu[Add0~62]cout</td>
</tr>
<tr>
<td>9.878</td>
<td>0.174</td>
<td>FF</td>
<td>IC 1</td>
<td>LCCOMB_X19_Y14_N14 theAlu[Add0~62]cout</td>
</tr>
<tr>
<td>10.336</td>
<td>0.458</td>
<td>FF</td>
<td>CELL 1</td>
<td>LCCOMB_X19_Y14_N16 theAlu[Add0~64]combout</td>
</tr>
<tr>
<td>11.260</td>
<td>0.924</td>
<td>FF</td>
<td>IC 1</td>
<td>LCCOMB_X19_Y18_N0 theAlu[O_out[31]~39]data</td>
</tr>
<tr>
<td>11.582</td>
<td>0.322</td>
<td>FF</td>
<td>CELL 12</td>
<td>LCCOMB_X19_Y18_N0 theAlu[O_out[31]~39]combout</td>
</tr>
<tr>
<td>12.517</td>
<td>0.935</td>
<td>FF</td>
<td>IC 1</td>
<td>LCCOMB_X20_Y15_N4 dataMem[Equal0~0]data</td>
</tr>
<tr>
<td>12.694</td>
<td>0.177</td>
<td>FR</td>
<td>CELL 1</td>
<td>LCCOMB_X20_Y15_N4 dataMem[Equal0~0]combout</td>
</tr>
<tr>
<td>13.004</td>
<td>0.310</td>
<td>RR</td>
<td>IC 1</td>
<td>LCCOMB_X20_Y15_N12 dataMem[Equal0~4]data</td>
</tr>
<tr>
<td>13.326</td>
<td>0.322</td>
<td>RR</td>
<td>CELL 31</td>
<td>LCCOMB_X20_Y15_N12 dataMem[Equal0~4]combout</td>
</tr>
<tr>
<td>14.793</td>
<td>0.178</td>
<td>RR</td>
<td>CELL 7</td>
<td>LCCOMB_X18_Y13_N4 muxWriteRegData[O_out[3]~3]combout</td>
</tr>
<tr>
<td>16.393</td>
<td>0.850</td>
<td>RR</td>
<td>IC 1</td>
<td>LCCOMB_X18_Y18_N24 muxWriteRegData[O_out[2]~2]combout</td>
</tr>
<tr>
<td>17.243</td>
<td>0.850</td>
<td>RR</td>
<td>IC 1</td>
<td>LCCOMB_X18_Y18_N0 rf[regs~8]data</td>
</tr>
<tr>
<td>17.421</td>
<td>0.178</td>
<td>RR</td>
<td>CELL 3</td>
<td>LCCOMB_X18_Y16_N0 rf[regs~8]combout</td>
</tr>
<tr>
<td>18.254</td>
<td>0.833</td>
<td>RR</td>
<td>IC 1</td>
<td>LCFF_X20_Y16_N27 rf[regs[0][2]]data</td>
</tr>
</tbody>
</table>
| 18.667  | 0.413 | RR   | CELL 1      | LCFF_X20_Y16_N27 register_file:rf[regs[0][2]]
<table>
<thead>
<tr>
<th>Time</th>
<th>Value</th>
<th>Type</th>
<th>ID</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.983</td>
<td>0.080</td>
<td>FR</td>
<td>CELL</td>
<td>LCCOMB_X19_Y15_N28 theAlu</td>
</tr>
<tr>
<td>9.144</td>
<td>0.161</td>
<td>RF</td>
<td>CE</td>
<td>LCCOMB_X19_Y15_N30 theAlu</td>
</tr>
<tr>
<td>9.144</td>
<td>0.000</td>
<td>FF</td>
<td>IC</td>
<td>LCCOMB_X19_Y15_N0  theAlu</td>
</tr>
<tr>
<td>9.224</td>
<td>0.080</td>
<td>RF</td>
<td>CELL</td>
<td>LCCOMB_X19_Y14_N0  theAlu</td>
</tr>
<tr>
<td>9.224</td>
<td>0.080</td>
<td>RR</td>
<td>IC</td>
<td>LCCOMB_X19_Y14_N2  theAlu</td>
</tr>
<tr>
<td>9.304</td>
<td>0.080</td>
<td>RF</td>
<td>CELL</td>
<td>LCCOMB_X19_Y14_N2  theAlu</td>
</tr>
<tr>
<td>9.384</td>
<td>0.080</td>
<td>RF</td>
<td>IC</td>
<td>LCCOMB_X19_Y14_N4  theAlu</td>
</tr>
<tr>
<td>9.464</td>
<td>0.080</td>
<td>RF</td>
<td>CELL</td>
<td>LCCOMB_X19_Y14_N6  theAlu</td>
</tr>
<tr>
<td>9.544</td>
<td>0.080</td>
<td>RR</td>
<td>IC</td>
<td>LCCOMB_X19_Y14_N8  theAlu</td>
</tr>
<tr>
<td>9.704</td>
<td>0.080</td>
<td>FR</td>
<td>CELL</td>
<td>LCCOMB_X19_Y14_N10 theAlu</td>
</tr>
<tr>
<td>9.704</td>
<td>0.080</td>
<td>RR</td>
<td>IC</td>
<td>LCCOMB_X19_Y14_N12 theAlu</td>
</tr>
<tr>
<td>9.878</td>
<td>0.174</td>
<td>RF</td>
<td>IC</td>
<td>LCCOMB_X19_Y14_N14 theAlu</td>
</tr>
<tr>
<td>10.336</td>
<td>0.458</td>
<td>FF</td>
<td>CELL</td>
<td>LCCOMB_X19_Y14_N16 theAlu</td>
</tr>
<tr>
<td>11.260</td>
<td>0.924</td>
<td>FF</td>
<td>IC</td>
<td>LCCOMB_X19_Y18_N0  theAlu</td>
</tr>
<tr>
<td>11.582</td>
<td>0.322</td>
<td>RF</td>
<td>CE</td>
<td>LCCOMB_X20_Y15_N4  dataMem</td>
</tr>
<tr>
<td>12.517</td>
<td>0.935</td>
<td>FF</td>
<td>IC</td>
<td>LCCOMB_X20_Y15_N8  dataMem</td>
</tr>
<tr>
<td>12.694</td>
<td>0.177</td>
<td>FR</td>
<td>CELL</td>
<td>LCCOMB_X20_Y15_N12 dataMem</td>
</tr>
<tr>
<td>13.004</td>
<td>0.310</td>
<td>FF</td>
<td>IC</td>
<td>LCCOMB_X20_Y15_N14 dataMem</td>
</tr>
<tr>
<td>13.326</td>
<td>0.322</td>
<td>RR</td>
<td>CELL</td>
<td>LCCOMB_X20_Y15_N16 dataMem</td>
</tr>
<tr>
<td>14.615</td>
<td>1.289</td>
<td>RR</td>
<td>IC</td>
<td>LCCOMB_X18_Y13_N4  muxWriteRegData</td>
</tr>
<tr>
<td>14.793</td>
<td>0.178</td>
<td>RR</td>
<td>CELL</td>
<td>LCCOMB_X18_Y13_N4  muxWriteRegData</td>
</tr>
<tr>
<td>16.071</td>
<td>1.278</td>
<td>RR</td>
<td>IC</td>
<td>LCCOMB_X18_Y18_N24 muxWriteRegData</td>
</tr>
<tr>
<td>16.393</td>
<td>0.322</td>
<td>RR</td>
<td>CELL</td>
<td>LCCOMB_X18_Y18_N24 muxWriteRegData</td>
</tr>
<tr>
<td>17.243</td>
<td>0.850</td>
<td>RR</td>
<td>IC</td>
<td>LCCOMB_X18_Y16_N0  rf</td>
</tr>
<tr>
<td>17.421</td>
<td>0.178</td>
<td>RR</td>
<td>CELL</td>
<td>LCCOMB_X18_Y16_N0  rf</td>
</tr>
<tr>
<td>18.254</td>
<td>0.833</td>
<td>RR</td>
<td>IC</td>
<td>LCFF_X20_Y16_N27 rf</td>
</tr>
<tr>
<td>18.667</td>
<td>0.413</td>
<td>RR</td>
<td>CELL</td>
<td>LCFF_X20_Y16_N27 register_file:rf</td>
</tr>
<tr>
<td>Index</td>
<td>Value</td>
<td>Type</td>
<td>Stage</td>
<td>Address</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>------</td>
<td>-------</td>
<td>------------------</td>
</tr>
<tr>
<td>8</td>
<td>983</td>
<td>FR</td>
<td>CELL</td>
<td>LCCOMB_X19_Y15_N28</td>
</tr>
<tr>
<td></td>
<td>983</td>
<td>RR</td>
<td>IC</td>
<td>LCCOMB_X19_Y15_N30</td>
</tr>
<tr>
<td>9</td>
<td>144</td>
<td>RF</td>
<td>CELL</td>
<td>LCCOMB_X19_Y15_N30</td>
</tr>
<tr>
<td></td>
<td>144</td>
<td>RR</td>
<td>IC</td>
<td>LCCOMB_X19_Y14_N0</td>
</tr>
<tr>
<td>9.224</td>
<td>008</td>
<td>FR</td>
<td>CELL</td>
<td>LCCOMB_X19_Y14_N0</td>
</tr>
<tr>
<td>9.224</td>
<td>000</td>
<td>RR</td>
<td>IC</td>
<td>LCCOMB_X19_Y14_N0</td>
</tr>
<tr>
<td>9</td>
<td>034</td>
<td>RR</td>
<td>IC</td>
<td>LCCOMB_X19_Y14_N2</td>
</tr>
<tr>
<td></td>
<td>034</td>
<td>FF</td>
<td>IC</td>
<td>LCCOMB_X19_Y14_N4</td>
</tr>
<tr>
<td>9</td>
<td>084</td>
<td>RR</td>
<td>IC</td>
<td>LCCOMB_X19_Y14_N4</td>
</tr>
<tr>
<td></td>
<td>084</td>
<td>RR</td>
<td>IC</td>
<td>LCCOMB_X19_Y14_N4</td>
</tr>
<tr>
<td></td>
<td>084</td>
<td>FF</td>
<td>IC</td>
<td>LCCOMB_X19_Y14_N6</td>
</tr>
<tr>
<td>9.464</td>
<td>080</td>
<td>FR</td>
<td>CELL</td>
<td>LCCOMB_X19_Y14_N8</td>
</tr>
<tr>
<td>9.464</td>
<td>080</td>
<td>RR</td>
<td>IC</td>
<td>LCCOMB_X19_Y14_N8</td>
</tr>
<tr>
<td>9.544</td>
<td>080</td>
<td>FF</td>
<td>IC</td>
<td>LCCOMB_X19_Y14_N10</td>
</tr>
<tr>
<td>9.624</td>
<td>080</td>
<td>FR</td>
<td>CELL</td>
<td>LCCOMB_X19_Y14_N10</td>
</tr>
<tr>
<td>9.624</td>
<td>080</td>
<td>RR</td>
<td>IC</td>
<td>LCCOMB_X19_Y14_N10</td>
</tr>
<tr>
<td>9.704</td>
<td>080</td>
<td>FR</td>
<td>CELL</td>
<td>LCCOMB_X19_Y14_N12</td>
</tr>
<tr>
<td>9.704</td>
<td>080</td>
<td>RR</td>
<td>IC</td>
<td>LCCOMB_X19_Y14_N12</td>
</tr>
<tr>
<td>9.878</td>
<td>080</td>
<td>RF</td>
<td>CELL</td>
<td>LCCOMB_X19_Y14_N14</td>
</tr>
<tr>
<td>9.878</td>
<td>080</td>
<td>FF</td>
<td>IC</td>
<td>LCCOMB_X19_Y14_N14</td>
</tr>
<tr>
<td>10.336</td>
<td>458</td>
<td>FF</td>
<td>CELL</td>
<td>LCCOMB_X19_Y14_N16</td>
</tr>
<tr>
<td>11.260</td>
<td>924</td>
<td>FF</td>
<td>IC</td>
<td>LCCOMB_X19_Y18_N0</td>
</tr>
<tr>
<td>11.582</td>
<td>322</td>
<td>FF</td>
<td>CELL</td>
<td>LCCOMB_X19_Y18_N0</td>
</tr>
<tr>
<td>12.517</td>
<td>935</td>
<td>FF</td>
<td>IC</td>
<td>LCCOMB_X20_Y15_N4</td>
</tr>
<tr>
<td>12.694</td>
<td>177</td>
<td>FF</td>
<td>CELL</td>
<td>LCCOMB_X20_Y15_N4</td>
</tr>
<tr>
<td>13.004</td>
<td>310</td>
<td>FF</td>
<td>CELL</td>
<td>LCCOMB_X20_Y15_N12</td>
</tr>
<tr>
<td>13.326</td>
<td>322</td>
<td>RR</td>
<td>CELL</td>
<td>LCCOMB_X20_Y15_N12</td>
</tr>
<tr>
<td>14.615</td>
<td>1289</td>
<td>RR</td>
<td>IC</td>
<td>LCCOMB_X18_Y13_N4</td>
</tr>
<tr>
<td>14.793</td>
<td>178</td>
<td>RR</td>
<td>CELL</td>
<td>LCCOMB_X18_Y13_N4</td>
</tr>
<tr>
<td>16.071</td>
<td>1278</td>
<td>RR</td>
<td>IC</td>
<td>LCCOMB_X19_Y18_N24</td>
</tr>
<tr>
<td>16.393</td>
<td>322</td>
<td>RR</td>
<td>CELL</td>
<td>LCCOMB_X19_Y18_N24</td>
</tr>
<tr>
<td>17.243</td>
<td>850</td>
<td>RR</td>
<td>IC</td>
<td>LCCOMB_X18_Y16_N0</td>
</tr>
<tr>
<td>17.421</td>
<td>178</td>
<td>RR</td>
<td>CELL</td>
<td>LCCOMB_X18_Y16_N0</td>
</tr>
<tr>
<td>18.254</td>
<td>833</td>
<td>RR</td>
<td>IC</td>
<td>LCCOMB_X20_Y16_N27</td>
</tr>
<tr>
<td>18.667</td>
<td>413</td>
<td>RR</td>
<td>CELL</td>
<td>LCCOMB_X20_Y16_N27</td>
</tr>
<tr>
<td>Index</td>
<td>Value</td>
<td>Type</td>
<td>ID</td>
<td>Comment</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>------</td>
<td>------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>1</td>
<td>2.850</td>
<td>RR</td>
<td>LCCOMB_X18_Y16_N0</td>
<td>rf[reg~8]dataad</td>
</tr>
<tr>
<td>3</td>
<td>1.783</td>
<td>RR</td>
<td>LCCOMB_X18_Y16_N0</td>
<td>rf[reg~8]combout</td>
</tr>
<tr>
<td>4</td>
<td>0.833</td>
<td>RR</td>
<td>LCFF_X20_Y16_N27</td>
<td>rf[reg][2][2]idata</td>
</tr>
<tr>
<td>5</td>
<td>0.413</td>
<td>RR</td>
<td>CELL 1</td>
<td>LCCOMB_X20_Y16_N27</td>
</tr>
</tbody>
</table>
Single-cycle Implementation to scale

18.667ns -> 3.733ns == 80% reduction in CT

Lold = IC * CPI * CTold

Lnew = IC * CPI * CTnew

CTnew = 0.2 * CTold

Lnew = 0.2 * Lold

Speed up = Lold/Lnew = 5x
Single-cycle Implementation to scale

Ideal 5-stage Pipeline (3.733ns -> 267Mhz)

Realistic 5-stage Pipeline

- Pipe stages are imbalanced
- Longest == 6.5ns
- Shortest == 0.79ns
- Speedup = 18.667/6.5 == 2.8x
**Question 1: True/False**

The ideal CPI for a pipelined CPU is larger than for a non-pipelined CPU.

<table>
<thead>
<tr>
<th>Correct Answers</th>
<th>Percent Answered</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>13.433%</td>
</tr>
<tr>
<td>False</td>
<td>86.567%</td>
</tr>
</tbody>
</table>

Unanswered 0%
### Question 2: Multiple Answer

Which of the following can lead to imperfect pipelining?

<table>
<thead>
<tr>
<th>Correct Answers</th>
<th>Percent Correct</th>
<th>Percent Incorrect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction mix.</td>
<td>53.731%</td>
<td>46.269%</td>
</tr>
<tr>
<td>Pipeline register setup time.</td>
<td>83.582%</td>
<td>16.418%</td>
</tr>
<tr>
<td>Compiler optimizations.</td>
<td>92.537%</td>
<td>7.463%</td>
</tr>
<tr>
<td>Complex, long-latency datapath components.</td>
<td>89.552%</td>
<td>10.448%</td>
</tr>
</tbody>
</table>
Question 3: Multiple Answer

Which of the following is true of VLIW processors?

Correct Answers

- Each instruction word contains multiple instructions.  
- The instructions in an instruction word execute simultaneously.
- VLIW machines are challenging for compilers to utilize effectively.
- VLIW machines have been successful in some commercial domains.

Average Score 3.53731 points

<table>
<thead>
<tr>
<th>Correct Answers</th>
<th>Percent Correct</th>
<th>Percent Incorrect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each instruction word contains multiple instructions.</td>
<td>92.537%</td>
<td>7.463%</td>
</tr>
<tr>
<td>The instructions in an instruction word execute simultaneously.</td>
<td>92.537%</td>
<td>7.463%</td>
</tr>
<tr>
<td>VLIW machines are challenging for compilers to utilize effectively.</td>
<td>92.537%</td>
<td>7.463%</td>
</tr>
<tr>
<td>VLIW machines have been successful in some commercial domains.</td>
<td>76.119%</td>
<td>23.881%</td>
</tr>
</tbody>
</table>
Question 4:  Multiple Choice

Which of the following is true about CISC instruction sets?

Correct

- It is prohibitively expensive to build fast CISC processors.  
- CISC instructions sets are more complex than RISC instruction sets.  
- RISC instruction sets are more complex than CISC instruction sets.  
- Based on their use in today's devices, it appears that RISC instruction sets are better suited to mobile computing than CISC ISAs.
- Both 1 and 4.
- Both 2 and 4.

Percent Answered

- 0%  
- 2.985%  
- 5.97%  
- 1.493%  
- 2.985%  
- 86.567%  
- 0%

Unanswered
Pipelining is Tricky

• Simple pipelining is easy
  • If the data flows in one direction only
  • If the stages are independent
  • In fact the tool can do this automatically via “retiming” (If you are curious, experiment with this in Quartus).

• Not so, for processors.
  • Branch instructions affect the next PC -- backward flow
  • Instructions need values computed by previous instructions -- not independent
Not just tricky, Hazardous!

- Hazards are situations where pipelining does not work as elegantly as we would like
- Three kinds
  - Structural hazards -- we have run out of a hardware resource.
  - Data hazards -- an input is not available on the cycle it is needed.
  - Control hazards -- the next instruction is not known.
- Dealing with hazards increases complexity or decreases performance (or both)
- Dealing efficiently with hazards is much of what makes processor design hard.
  - That, and the Quartus tools ;-)
Hazards: Key Points

• Hazards cause imperfect pipelining
  • They prevent us from achieving CPI = 1
  • They are generally caused by “counter flow” data dependences in the pipeline

• Three kinds
  • Structural -- contention for hardware resources
  • Data -- a data value is not available when/where it is needed.
  • Control -- the next instruction to execute is not known.

• Two ways to deal with hazards
  • Removal -- add hardware and/or complexity to work around the hazard so it does not occur
  • Stall -- Hold up the execution of new instructions. Let the older instructions finish, so the hazard will clear.
A Structural Hazard

- Both the decode and write back stage have to access the register file.
- There is only one registers file. A structural hazard!!
- Solution: Write early, read late
  - Writes occur at the clock edge and complete long before the end of the cycle
  - This leave enough time for the outputs to settle for the reads.

- Hazard avoided!
Data Dependences

- A data dependence occurs whenever one instruction needs a value produced by another.
  - Register values
  - Also memory accesses (more on this later)

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
add $t3, $s0, $t4
add $t3, $t2, $t4
```
Data Dependences

• A data dependence occurs whenever one instruction needs a value produced by another.
  • Register values
  • Also memory accesses (more on this later)

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
add $t3, $s0, $t4
add $t3, $t2, $t4
```
Data Dependences

• A data dependence occurs whenever one instruction needs a value produced by another.
  • Register values
  • Also memory accesses (more on this later)

\[
\begin{align*}
\text{add} & \quad s0, \quad t0, \quad t1 \\
\text{sub} & \quad t2, \quad s0, \quad t3 \\
\text{add} & \quad t3, \quad s0, \quad t4 \\
\text{add} & \quad t3, \quad t2, \quad t4
\end{align*}
\]
Data Dependences

• A data dependence occurs whenever one instruction needs a value produced by another.
  • Register values
  • Also memory accesses (more on this later)

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
add $t3, $s0, $t4
add $t3, $t2, $t4
```
Data Dependences

- A data dependence occurs whenever one instruction needs a value produced by another.
  - Register values
  - Also memory accesses (more on this later)

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
add $t3, $s0, $t4
add $t3, $t2, $t4
```
Data Dependences

• A data dependence occurs whenever one instruction needs a value produced by another.
  • Register values
  • Also memory accesses (more on this later)

```assembly
add $s0, $t0, $t1
sub $t2, $s0, $t3
add $t3, $s0, $t4
add $t3, $t2, $t4
```
Dependences in the pipeline

- In our simple pipeline, these instructions cause a data hazard

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```
Dependences in the pipeline

- In our simple pipeline, these instructions cause a data hazard

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```
Dependences in the pipeline

- In our simple pipeline, these instructions cause a data hazard

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```
How can we fix it?

• Ideas?
Solution 1: Make the compiler deal with it.

- Expose hazards to the big A architecture
  - A result is available N instructions after the instruction that generates it.
  - In the meantime, the register file has the old value.
  - This is called “a register delay slot”
- What is N? Can it change?

```
add $s0, $t0, $t1
```

```
delay slot
```

```
delay slot
```

```
sub $t2, $s0, $t3
```
Solution 1: Make the compiler deal with it.

- Expose hazards to the big A architecture
  - A result is available $N$ instructions after the instruction that generates it.
  - In the meantime, the register file has the old value.
  - This is called “a register delay slot”
- What is $N$? Can it change? $N = 2$, for our design

```
add $s0, $t0, $t1

delay slot

sub $t2, $s0, $t3
```
Compiling for delay slots

- The compiler must fill the delay slots
- Ideally, with useful instructions, but nops will work too.
Solution 2: Stall

- When you need a value that is not ready, “stall”
  - Suspend the execution of the executing instruction
  - and those that follow.
  - This introduces a pipeline “bubble.”
- A bubble is a lack of work to do, it propagates through the pipeline like nop instructions

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
add $t3, $s0, $t4
```
Solution 2: Stall

• When you need a value that is not ready, “stall”
  • Suspend the execution of the executing instruction
  • and those that follow.
  • This introduces a pipeline “bubble.”
• A bubble is a lack of work to do, it propagates through the pipeline like nop instructions
Solution 2: Stall

- When you need a value that is not ready, “stall”
  - Suspend the execution of the executing instruction
  - and those that follow.
  - This introduces a pipeline “bubble.”
- A bubble is a lack of work to do, it propagates through the pipeline like nop instructions
Stalling the pipeline

- Freeze all pipeline stages before the stage where the hazard occurred.
  - Disable the PC update
  - Disable the pipeline registers
- This is equivalent to inserting into the pipeline when a hazard exists
  - Insert nop control bits at stalled stage (decode in our example)
  - How is this solution still potentially “better” than relying on the compiler?
Stalling the pipeline

- Freeze all pipeline stages before the stage where the hazard occurred.
  - Disable the PC update
  - Disable the pipeline registers
- This is equivalent to inserting into the pipeline when a hazard exists
  - Insert nop control bits at stalled stage (decode in our example)
  - How is this solution still potentially “better” than relying on the compiler?

The compiler can still act like there are delay slots to avoid stalls. Implementation details are not exposed in the ISA.
Calculating CPI for Stalls

- In this case, the bubble lasts for 2 cycles.
- As a result, in cycle (6 and 7), no instruction completes.
- What happens to CPI?
  - In the absence of stalls, CPI is one, since one instruction completes per cycle
  - If an instruction stalls for N cycles, it’s CPI goes up by N
Hardware for Stalling

- Turn off the enables on the earlier pipeline stages
  - The earlier stages will keep processing the same instruction over and over.
  - No new instructions get fetched.
- Insert control and data values corresponding to a nop into the “downstream” pipeline register.
  - This will create the bubble.
  - The nops will flow downstream, doing nothing.
- When the stall is over, re-enable the pipeline registers
  - The instructions in the “upstream” stages will start moving again.
  - New instructions will start entering the pipeline again.
The Impact of Stalling On Performance

• \( ET = I \times CPI \times CT \)
• \( I \) and \( CT \) are constant
• What is the impact of stalling on CPI?

• What do we need to know to figure it out?
The Impact of Stalling On Performance

- ET = I * CPI * CT
- I and CT are constant
- What is the impact of stalling on CPI?

- Fraction of instructions that stall: 30%
- Baseline CPI = 1
- Stall CPI = 1 + 2 = 3

- New CPI =
The Impact of Stalling On Performance

• ET = I * CPI * CT
• I and CT are constant
• What is the impact of stalling on CPI?

• Fraction of instructions that stall: 30%
• Baseline CPI = 1
• Stall CPI = 1 + 2 = 3

• New CPI = $0.3 \times 3 + 0.7 \times 1 = 1.6$
Solution 3: Bypassing/Forwarding

- Data values are computed in Ex and Mem but “publicized in write back”

- The data exists! We should use it.

![Diagram showing the flow of operations with inputs, results known, and results "published" to registers.]
Bypassing or Forwarding

• Take the values, where ever they are

\[
\begin{align*}
\text{add} & \quad s0, \ t0, \ t1 \\
\text{sub} & \quad t2, \ s0, \ t3
\end{align*}
\]
Forwarding Paths

- add $s0, $t0, $t1
- sub $t2, $s0, $t3
- sub $t2, $s0, $t3
- sub $t2, $s0, $t3
Forwarding in Hardware

Diagram showing the flow of data through different components such as Instruction Memory, Register File, Data Memory, ALU, and various processing stages like IFetch/Dec, Dec/Exec, Exec/Mem, and Mem/WB. Blocks and arrows highlight the flow of addresses and data through these stages.
Hardware Cost of Forwarding

• In our pipeline, adding forwarding required relatively little hardware.
• For deeper pipelines it gets much more expensive
  • Roughly: ALU * pipe_stages you need to forward over
  • Some modern processor have multiple ALUs (4-5)
  • And deeper pipelines (4-5 stages of to forward across)
• Not all forwarding paths need to be supported.
  • If a path does not exist, the processor will need to stall.
Forwarding for Loads

- Values can also come from the Mem stage
  - In this case, forward is not possible to the next instruction (and is not necessary for later instructions)

- Choices
  - Always stall!
  - Stall when needed!
  - Expose this in the ISA.
Forwarding for Loads

- Values can also come from the Mem stage
  - In this case, forward is not possible to the next instruction (and is not necessary for later instructions)

- Choices
  - Always stall!
  - Stall when needed!
  - Expose this in the ISA.

Time travel presents significant implementation challenges
Forwarding for Loads

• Values can also come from the Mem stage
  • In this case, forward is not possible to the next instruction (and is not necessary for later instructions)

• Choices
  • Always stall!
  • Stall when needed!
  • Expose this in the ISA.

Which does MIPS do?

Time travel presents significant implementation challenges
Pros and Cons

• Punt to the compiler
  • This is what MIPS does and is the source of the load-delay slot
  • Future versions must emulate a single load-delay slot.
  • The compiler fills the slot if possible, or drops in a nop.

• Always stall.
  • The compiler is oblivious, but performance will suffer
  • 10-15% of instructions are loads, and the CPI for loads will be 2

• Forward when possible, stall otherwise
  • Here the compiler can order instructions to avoid the stall.
  • If the compiler can’t fix it, the hardware will.
Stalling for Load

Load $s1, 0($s1)

Addi $t1, $s1, 4

To “stall” we insert a noop in place of the instruction and freeze the earlier stages of the pipeline
To "stall" we insert a noop *in place of* the instruction and freeze the earlier stages of the pipeline.
To “stall” we insert a noop *in place of* the instruction and freeze the earlier stages of the pipeline.
Inserting Noops

To “stall” we insert a noop in place of the instruction and freeze the earlier stages of the pipeline.

Load $s1, 0($s1)

Addi $t1, $s1, 4

The noop is in Mem

Noop inserted

The noop is in Mem
Key Points: Control Hazards

- Control occur when we don’t know what the next instruction is
- Caused by branches and jumps.
- Strategies for dealing with them
  - Stall
  - Guess!
    - Leads to speculation
    - Flushing the pipeline
    - Strategies for making better guesses
- Understand the difference between stall and flush
Computing the PC Normally

- Non-branch instruction
  - $PC = PC + 4$
- When is PC ready?

Diagram:

1. Fetch
2. Decode
3. EX
4. Mem
5. Write back
Computing the PC Normally

• Non-branch instruction
  • $PC = PC + 4$
• When is PC ready?

Fetch → Decode → EX → Mem → Write back
Computing the PC Normally

- Non-branch instruction
  - $PC = PC + 4$
- When is PC ready?

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
sub $t2, $s0, $t3
sub $t2, $s0, $t3
```
Fixing the Ubiquitous Control Hazard

• We need to know if an instruction is a branch in the fetch stage!
• How can we accomplish this?

Solution 1: Partially decode the instruction in fetch. You just need to know if it’s a branch, a jump, or something else.

Solution 2: We’ll discuss later.
Fixing the Ubiquitous Control Hazard

- We need to know if an instruction is a branch in the fetch stage!
- How can we accomplish this?

**Solution 1:** Partially decode the instruction in fetch. You just need to know if it’s a branch, a jump, or something else.

**Solution 2:** We’ll discuss later.
Computing the PC Normally

- Pre-decode in the fetch unit.
  - \( \text{PC} = \text{PC} + 4 \)
- The PC is ready for the next fetch cycle.
Computing the PC Normally

- Pre-decode in the fetch unit.
  - $\text{PC} = \text{PC} + 4$
- The PC is ready for the next fetch cycle.
Computing the PC Normally

- Pre-decode in the fetch unit.
  - \( PC = PC + 4 \)
- The PC is ready for the next fetch cycle.

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
sub $t2, $s0, $t3
sub $t2, $s0, $t3
```
Computing the PC for Branches

- **Branch instructions**
  - `bne $s1, $s2, offset`
  - `if ($s1 != $s2) { PC = PC + offset} else {PC = PC + 4;}`
- **When is the value ready?**

\[\text{sll } $s4, $t6, $t5\]
\[\text{bne } $t2, $s0, \text{somewhere}\]
\[\text{add } $s0, $t0, $t1\]
\[\text{and } $s4, $t0, $t1\]
Computing the PC for Branches

- Branch instructions
  - `bne $s1, $s2, offset`
  - `if ($s1 != $s2) { PC = PC + offset} else {PC = PC + 4;}`
- When is the value ready?
Computing the PC for Jumps

- Jump instructions
  - `jr $s1` -- jump register
  - `PC = $s1`
- When is the value ready?

```
sll $s4, $t6, $t5
jr $s4
add $s0, $t0, $t1
```

Cycles
Computing the PC for Jumps

- Jump instructions
  - `jr $s1` -- jump register
  - `PC = $s1`
- When is the value ready?

```
sll $s4, $t6, $t5
jr $s4
add $s0, $t0, $t1
```
Dealing with Branches: Option 0 -- stall

- What does this do to our CPI?

```
sll $s4, $t6, $t5
bne $t2, $s0, somewhere
add $s0, $t0, $t1
and $s4, $t0, $t1
```
Option 1: The compiler

- Use “branch delay” slots.
- The next N instructions after a branch are always executed
- How big is N?
  - For jumps?
  - For branches?
- Good
  - Simple hardware
- Bad
  - N cannot change.
Delays slots.

- **Taken**
  - `bne $t2, $s0, somewhere`
  - `add $t2, $s4, $t1`
  - `add $s0, $t0, $t1`
  - `...`
  - `somewhere: sub $t2, $s0, $t3`

- **Branch Delay**
But MIPS Only Has One Delay Slot!

- The second branch delay slot is expensive!
  - Filling one slot is hard. Filling two is even more so.
- Solution!: Resolve branches in decode.
For the rest of this slide deck, we will assume that MIPS has no branch delay slot.

If you have questions about whether part of the homework/test/quiz makes this assumption ask or make it clear what you assumed.
Option 2: Simple Prediction

- Can a processor tell the future?
- For non-taken branches, the new PC is ready immediately.
- Let’s just assume the branch is not taken
- Also called “branch prediction” or “control speculation”
- What if we are wrong?
- Branch prediction vocabulary
  - Prediction -- a guess about whether a branch will be taken or not taken
  - Misprediction -- a prediction that turns out to be incorrect.
  - Misprediction rate -- fraction of predictions that are incorrect.
• We start the add, and then, when we discover the branch outcome, we *squash* it.

• Also called “flushing the pipeline”

• Just like a stall, flushing one instruction increases the branch’s CPI by 1
When we flush the pipeline, we convert instructions into noops
- Turn off the write enables for write back and mem stages
- Disable branches (i.e., make sure the ALU does raise the branch signal).

Instructions do not stop moving through the pipeline

For the example on the previous slide the “inject_nop_decode_execute” signal will go high for one cycle.
Flush the Pipeline

- When we flush the pipe, we convert instructions into noops
  - Turn off the write enables for write back and mem stages
  - Disable branches (i.e., make sure the ALU does raise the branch signal).
- Instructions *do not stop* moving through the pipeline
- For the example on the previous slide the “inject_nop_decode_execute” signal will go high for one cycle.

These signals for *stalling*
Flushing the Pipeline

• When we flush the pipe, we convert instructions into noops
  • Turn off the write enables for write back and mem stages
  • Disable branches (i.e., make sure the ALU does raise the branch signal).

• Instructions *do not stop* moving through the pipeline

• For the example on the previous slide the “inject_nop_decode_execute” signal will go high for one cycle.

These signals for *stalling*

This signal is for both stalling and flushing

Control signals
Simple “static” Prediction

• “static” means before run time
• Many prediction schemes are possible
• Predict taken
  • Pros?
• Predict not-taken
  • Pros?
• Backward taken/Forward not taken
  • The best of both worlds!
  • Most loops have have a backward branch at the bottom, those will predict taken
  • Others (non-loop) branches will be not-taken.
Simple “static” Prediction

- “static” means before run time
- Many prediction schemes are possible
- Predict taken
  - Pros?
- Predict not-taken
  - Pros?
- Backward taken/Forward not taken
  - The best of both worlds!
  - Most loops have have a backward branch at the bottom, those will predict taken
  - Others (non-loop) branches will be not-taken.

Loops are commons
Simple “static” Prediction

- “static” means before run time
- Many prediction schemes are possible
- Predict taken
  - Pros? Loops are commons
- Predict not-taken
  - Pros? Not all branches are for loops.
- Backward taken/Forward not taken
  - The best of both worlds!
  - Most loops have have a backward branch at the bottom, those will predict taken
  - Others (non-loop) branches will be not-taken.
Basic Pipeline Recap

- The PC is required in Fetch
- For branches, it’s not know till *decode*.

Branches only, one delay slot, simplified ISA, no control
• Predict: Compute all possible next PCs in fetch. Choose one.
  • The correct next PC is known in decode
• Flush as needed: Replace "wrong path" instructions with no-ops.
• Predict: Compute all possible next PCs in fetch. Choose one.
• The correct next PC is known in decode
• Flush as needed: Replace “wrong path” instructions with no-ops.
• Predict: Compute all possible next PCs in fetch. Choose one.
  • The correct next PC is known in decode
• Flush as needed: Replace “wrong path” instructions with no-ops.
• Predict: Compute all possible next PCs in fetch. Choose one.
  • The correct next PC is known in decode
• Flush as needed: Replace “wrong path” instructions with no-ops.
Supporting Speculation

- **Predict**: Compute all possible next PCs in fetch. Choose one.
- **The correct next PC is known in decode**
- **Flush as needed**: Replace “wrong path” instructions with no-ops.
Supporting Speculation

Squash/Flush Control

- Predict: Compute all possible next PCs in fetch. Choose one.
- The correct next PC is known in decode
- Flush as needed: Replace “wrong path” instructions with no-ops.
Implementing Backward taken/forward not taken (BTFNT)

- A new “branch predictor” module determines what guess we are going to make.
- The BTFNT branch predictor has one input
  - The sign of the offset -- to make the prediction
  - The branch signal from the comparator -- to check if the prediction was correct.
- And two output
  - The PC mux selector
    - Steers execution in the predicted direction
    - Re-directs execution when the branch resolves.
  - A mis-predict signal that causes control to flush the pipe.
Performance Impact (ex 1)

- $ET = I \times CPI \times CT$

- BTFTN is has a misprediction rate of 20%.
- Branches are 20% of instructions
- Changing the front end increases the cycle time by 10%
- What is the speedup BTFNT compared to just stalling on every branch?
Performance Impact (ex 1)

- ET = I * CPI * CT

- Back taken, forward not taken is 80% accurate
- Branches are 20% of instructions
- Changing the front end increases the cycle time by 10%
- What is the speedup Bt/Fnt compared to just stalling on every branch?
- Btfnt
  - CPI = 0.2*0.2*(1 + 1) + (1-.2*.2)*1 = 1.04
  - CT = 1.1
  - IC = IC
  - ET = 1.144

- Stall
  - CPI = .2*2 + .8*1 = 1.2
  - CT = 1
  - IC = IC
  - ET = 1.2

- Speed up = 1.2/1.144 = 1.05
The Branch Delay Penalty

• The number of cycle between fetch and branch resolution is called the “branch delay penalty”
  • It is the number of instruction that get flushed on a misprediction.
  • It is the number of extra cycles the branch gets charged (i.e., the CPI for mispredicted branches goes up by the penalty for)
Performance Impact (ex 2)

- ET = I * CPI * CT
- Our current design resolves branches in decode, so the branch delay penalty is 1 cycle.
- If removing the comparator from decode (and resolving branches in execute) would reduce cycle time by 20%, would it help or hurt performance?
  - Mis predict rate = 20%
  - Branches are 20% of instructions

Resolve in Decode
- CPI = 0.2*0.2*(1 + 1) + (1-.2*.2)*1 = 1.04
- CT = 1
- IC = IC
- ET = 1.04

Resolve in execute
- CPI = 0.2*0.2*(1 + 2) + (1-.2*.2)*1 = 1.08
- CT = 0.8
- IC = IC
- ET = 0.864

- Speedup = 1.2
Performance Impact (ex 2)

• ET = I * CPI * CT
• Our current design resolves branches in decode, so the branch delay penalty is 1 cycle.
• If removing the comparator from decode (and resolving branches in execute) would reduce cycle time by 20%, would it help or hurt performance?
  • Mis predict rate = 20%
  • Branches are 20% of instructions
The Importance of Pipeline depth

• There are two important parameters of the pipeline that determine the impact of branches on performance
  • Branch decode time -- how many cycles does it take to identify a branch (in our case, this is less than 1)
  • Branch resolution time -- cycles until the real branch outcome is known (in our case, this is 2 cycles)
Pentium 4 pipeline

- Branches take 19 cycles to resolve
- Identifying a branch takes 4 cycles.
- Stalling is not an option.
- 80% branch prediction accuracy is also not an option.
- Not quite as bad now, but BP is still very important.
Performance Impact (ex 1)

- \( ET = I \cdot CPI \cdot CT \)

- Back taken, forward not taken is 80% accurate
- Branches are 20% of instructions
- Changing the front end increases the cycle time by 10%
- What is the speedup \( Bt/Fnt \) compared to just stalling on every branch?

By taking back:
- \( CPI = 0.2 \cdot 0.2 \cdot (1 + 1) + (1 - 0.2 \cdot 0.2) \cdot 1 = 1.04 \)
- \( CT = 1.144 \)
- \( IC = IC \)
- \( ET = 1.144 \)

By stalling:
- \( CPI = 0.2 \cdot 2 + 0.8 \cdot 1 = 1.2 \)
- \( CT = 1 \)
- \( IC = IC \)
- \( ET = 1.2 \)

Speed up = \( \frac{1.2}{1.144} = 1.05 \)

What if this were 20 instead of 1?
Performance Impact (ex 1)

- ET = I * CPI * CT

- Back taken, forward not taken is 80% accurate
- Branches are 20% of instructions
- Changing the front end increases the cycle time by 10%
- What is the speedup Bt/Fnt compared to just stalling on every branch?
  - Btfnt
    - CPI = 0.2*0.2*(1 + 1) + (1-.2*.2)*1 = 1.04
    - CT = 1.144
    - IC = IC
    - ET = 1.144

- Stall
  - CPI = .2*2 + .8*1 = 1.2
  - CT = 1
  - IC = IC
  - ET = 1.2

- Speed up = 1.2/1.144 = 1.05

What if this were 20 instead of 1?

Branches are relatively infrequent (~20% of instructions), but Amdahl’s Law tells that we can’t completely ignore this uncommon case.
Dynamic Branch Prediction

- Long pipes demand higher accuracy than static schemes can deliver.
- Instead of making the guess once (i.e. statically), make it every time we see the branch.
- Many ways to predict dynamically
  - We will focus on predicting future behavior based on past behavior
Predictable control

- Use previous branch behavior to predict future branch behavior.
- When is branch behavior predictable?
Predictable control

• Use previous branch behavior to predict future branch behavior.

• When is branch behavior predictable?
  • Loops -- for(i = 0; i < 10; i++) {}  9 taken branches, 1 not-taken branch. All 10 are pretty predictable.
  • Run-time constants
    • Foo(int v,) { for (i = 0; i < 1000; i++) {if (v) {...}}}
    • The branch is always taken or not taken.
  • Corollated control
    • a = 10;  b = <something usually larger than a >
    • if (a > 10) {}
    • if (b > 10) {}
  • Function calls
    • LibraryFunction() -- Converts to a jr (jump register) instruction, but it’s always the same.
    • BaseClass * t;  // t is usually a of sub class, SubClass
    • t->SomeVirtualFunction() // will usually call the same function
Dynamic Predictor 1: The Simplest Thing

• Predict that this branch will go the same way as the previous branch did.
• Pros?

• Cons?
Dynamic Predictor 1: The Simplest Thing

• Predict that this branch will go the same way as the previous branch did.

• Pros?

  Dead simple. Keep a bit in the fetch stage. Works ok for simple loops. The compiler might be able to arrange things to make it work better.

• Cons?
Dynamic Predictor 1: The Simplest Thing

- Predict that this branch will go the same way as the previous branch did.
- Pros?
  
  Dead simple. Keep a bit in the fetch stage. Works ok for simple loops. The compiler might be able to arrange things to make it work better.

- Cons?
  
  An unpredictable branch in a loop will mess everything up. It can’t tell the difference between branches
Dynamic Prediction 2: A table of bits

• Give each branch it’s own bit in a table
  • Look up the prediction bit for the branch
  • How big does the table need to be?

• Pros:

• Cons:
Dynamic Prediction 2: A table of bits

• Give each branch it’s own bit in a table
  • Look up the prediction bit for the branch
  • How big does the table need to be?

• Pros:
  It can differentiate between branches.
  Bad behavior by one won’t mess up others.... mostly.

• Cons:
Dynamic Prediction 2: A table of bits

• Give each branch it’s own bit in a table
  • Look up the prediction bit for the branch
  • How big does the table need to be?

  Infinite! Bigger is better, but don’t mess with the cycle time. Index into it using the low order bits of the PC

• Pros: It can differentiate between branches. Bad behavior by one won’t mess up others.... mostly.

• Cons:
Dynamic Prediction 2: A table of bits

- Give each branch it’s own bit in a table
  - Look up the prediction bit for the branch
  - How big does the table need to be?

  Infinite! Bigger is better, but don’t mess with the cycle time. Index into it using the low order bits of the PC

- Pros: It can differentiate between branches. Bad behavior by one won’t mess up others.... mostly.

- Cons: Accuracy is still not great.
Branch Prediction Trick #1

- Associating prediction state with a particular branch.
- We would like to keep separate prediction state for every *static* branch.
  - In practice this is not possible, since there are a potentially unbounded number of branches.
- Instead, we use a heuristic to associate prediction state with a branch.
  - The simplest heuristic is to use the low-order bits of the PC to select the prediction state.

<table>
<thead>
<tr>
<th>PC</th>
<th>Low order bits</th>
</tr>
</thead>
</table>

Table of predictor state

Prediction
Dynamic Prediction 2: A table of bits

```c
while (1) {
    for(j = 0; j < 4; j++) {
        // branch at address 0x100A
    }
}
```

<table>
<thead>
<tr>
<th>iteration</th>
<th>Actual</th>
<th>prediction</th>
<th>new prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>taken</td>
<td>not taken</td>
<td>taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
<td>taken</td>
<td>not taken</td>
</tr>
<tr>
<td>1</td>
<td>taken</td>
<td>not taken</td>
<td>take</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
</tbody>
</table>
Dynamic Prediction 2: A table of bits

while (1) {
    for(j = 0; j < 4; j++) {
        // branch at address 0x100A
    }
}

<table>
<thead>
<tr>
<th>iteration</th>
<th>Actual</th>
<th>prediction</th>
<th>new prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>taken</td>
<td>not taken</td>
<td>taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
<td>taken</td>
<td>not taken</td>
</tr>
<tr>
<td>1</td>
<td>taken</td>
<td>not taken</td>
<td>take</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
</tbody>
</table>

Table 16 of "last direction bits"

0xA

• What’s the accuracy for the branch?
Dynamic Prediction 2: A table of bits

while (1) {
    for(j = 0; j < 4; j++) {
        // branch at address 0x100A
    }
}

<table>
<thead>
<tr>
<th>iteration</th>
<th>Actual</th>
<th>prediction</th>
<th>new prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>taken</td>
<td>not taken</td>
<td>taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
<td>taken</td>
<td>not taken</td>
</tr>
<tr>
<td>1</td>
<td>taken</td>
<td>not taken</td>
<td>take</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
</tbody>
</table>

Table 16 of "last direction bits"

0xA

- What’s the accuracy for the branch?
Dynamic Prediction 2: A table of bits

while (1) {
    for(j = 0; j < 4; j++) {
        // branch at address 0x100A
    }
}

<table>
<thead>
<tr>
<th>iteration</th>
<th>Actual</th>
<th>prediction</th>
<th>new prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>taken</td>
<td>not taken</td>
<td>taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
<td>taken</td>
<td>not taken</td>
</tr>
<tr>
<td>1</td>
<td>taken</td>
<td>not taken</td>
<td>take</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
</tbody>
</table>

Table 16 of "last direction bits"

0xA

0x100A

**What’s the accuracy for the branch?**
Dynamic Prediction 2: A table of bits

while (1) {
    for (j = 0; j < 4; j++) {
        // branch at address 0x100A
    }
}

<table>
<thead>
<tr>
<th>iteration</th>
<th>Actual</th>
<th>prediction</th>
<th>new prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>taken</td>
<td>not taken</td>
<td>taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
<td>taken</td>
<td>not taken</td>
</tr>
<tr>
<td>1</td>
<td>taken</td>
<td>not taken</td>
<td>take</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
</tbody>
</table>

Table 16 of "last direction bits"

0xA

• What’s the accuracy for the branch?
Dynamic Prediction 2: A table of bits

```c
while (1) {
    for(j = 0; j < 4; j++) {
        // branch at address 0x100A
    }
}
```

<table>
<thead>
<tr>
<th>iteration</th>
<th>Actual</th>
<th>prediction</th>
<th>new prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>taken</td>
<td>not taken</td>
<td>taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
<td>taken</td>
<td>not taken</td>
</tr>
<tr>
<td>1</td>
<td>taken</td>
<td>not taken</td>
<td>take</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
</tbody>
</table>

**Table 16 of "last direction bits"**

- **0xA**
  - Prediction

- **What’s the accuracy for the branch?**
Dynamic Prediction 2: A table of bits

```c
while (1) {
    for(j = 0; j < 4; j++) {
        // branch at address 0x100A
    }
}
```

<table>
<thead>
<tr>
<th>iteration</th>
<th>Actual</th>
<th>prediction</th>
<th>new prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>taken</td>
<td>not taken</td>
<td>taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
<td>taken</td>
<td>not taken</td>
</tr>
<tr>
<td>1</td>
<td>taken</td>
<td>not taken</td>
<td>take</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
</tbody>
</table>

Table 16 of "last direction bits"

- 0xA not taken

- What’s the accuracy for the branch?
Dynamic Prediction 2: A table of bits

```c
while (1) {
    for(j = 0; j < 4; j++) {
        // branch at address 0x100A
    }
}
```

<table>
<thead>
<tr>
<th>iteration</th>
<th>Actual</th>
<th>prediction</th>
<th>new prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>taken</td>
<td>not taken</td>
<td>taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
<td>taken</td>
<td>not taken</td>
</tr>
<tr>
<td>1</td>
<td>taken</td>
<td>not taken</td>
<td>take</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
</tbody>
</table>

Table 16 of "last direction bits"

```
0xA
```

- What’s the accuracy for the branch?
Dynamic Prediction 2: A table of bits

```c
while (1) {
    for (j = 0; j < 4; j++) { // branch at address 0x100A
    }
}
```

<table>
<thead>
<tr>
<th>iteration</th>
<th>Actual</th>
<th>prediction</th>
<th>new prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>taken</td>
<td>not taken</td>
<td>taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
<td>taken</td>
<td>not taken</td>
</tr>
<tr>
<td>1</td>
<td>taken</td>
<td>not taken</td>
<td>take</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
</tbody>
</table>

Table 16 of "last direction bits"

- What’s the accuracy for the branch?
Dynamic Prediction 2: A table of bits

while (1) {
    for(j = 0; j < 4; j++) {
        // branch at address 0x100A
    }
}

<table>
<thead>
<tr>
<th>iteration</th>
<th>Actual</th>
<th>prediction</th>
<th>new prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>taken</td>
<td>not taken</td>
<td>taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
<td>taken</td>
<td>not taken</td>
</tr>
<tr>
<td>1</td>
<td>taken</td>
<td>not taken</td>
<td>take</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
</tbody>
</table>

• What’s the accuracy for the branch?
Dynamic Prediction 2: A table of bits

```java
while (1) {
    for(j = 0; j < 4; j++) {
        // branch at address 0x100A
    }
}
```

<table>
<thead>
<tr>
<th>iteration</th>
<th>Actual</th>
<th>prediction</th>
<th>new prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>taken</td>
<td>not taken</td>
<td>taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
<td>taken</td>
<td>not taken</td>
</tr>
<tr>
<td>1</td>
<td>taken</td>
<td>not taken</td>
<td>take</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
</tbody>
</table>

Table 16 of "last direction bits"

50% or 2 per loop

• What’s the accuracy for the branch?
Quiz 6

1. True/False
   a. Squashing instructions and stalling both result in increased CPI.
   b. Squashing can be used to resolve control hazards.
   c. Stalling cannot be used to resolve control hazards.
   d. Static branch predictors use tables of 2-bit counters.

2. Briefly explain why resolving branches in decode is necessary in the MIPS 5-stage pipeline (assuming it does not do branch prediction, and assuming MIPS has one delay slot).

3. Give two examples of why branch behavior is often predictable.

4. If we double the number of pipeline stages in the MIPS 5-stage design by dividing each existing stage in half, what would the new branch delay penalty be (assume branches resolve at the end of decode)?

5. On a scale of 1-10 (1 being completely unfair and 10 being completely fair), how fair was the midterm?
Dynamic prediction 3: A table of counters

• Instead of a single bit, keep two. This gives four possible states

• Taken branches move the state to the right. Not-taken branches move it to the left.

• The predictor waits one prediction before it changes its prediction
Dynamic Prediction 3: A table of counters

```c
for(i = 0; i < 10; i++) {
    for(j = 0; j < 4; j++) {
    }
}
```

What’s the accuracy for the inner loop’s branch? (start in weakly taken)
Dynamic Prediction 3: A table of counters

```plaintext
for(i = 0; i < 10; i++) {
    for(j = 0; j < 4; j++) {
    }
}
```

<table>
<thead>
<tr>
<th>iteration</th>
<th>Actual</th>
<th>state</th>
<th>prediction</th>
<th>new state</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>taken</td>
<td>weakly taken</td>
<td>taken</td>
<td>strongly taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>strongly taken</td>
<td>taken</td>
<td>strongly taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>strongly taken</td>
<td>taken</td>
<td>strongly taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
<td>strongly taken</td>
<td>taken</td>
<td>weakly taken</td>
</tr>
<tr>
<td>1</td>
<td>taken</td>
<td>weakly taken</td>
<td>taken</td>
<td>strongly taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>strongly taken</td>
<td>taken</td>
<td>strongly taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>strongly taken</td>
<td>taken</td>
<td>strongly taken</td>
</tr>
</tbody>
</table>

What’s the accuracy for the inner loop’s branch? (start in weakly taken)
Dynamic Prediction 3: A table of counters

```c
for (i = 0; i < 10; i++) {
    for (j = 0; j < 4; j++) {
    }
}
```

<table>
<thead>
<tr>
<th>iteration</th>
<th>Actual</th>
<th>state</th>
<th>prediction</th>
<th>new state</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>taken</td>
<td>weakly taken</td>
<td>taken</td>
<td>strongly taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>strongly taken</td>
<td>taken</td>
<td>strongly taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>strongly taken</td>
<td>taken</td>
<td>strongly taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
<td>strongly taken</td>
<td>taken</td>
<td>weakly taken</td>
</tr>
<tr>
<td>1</td>
<td>taken</td>
<td>weakly taken</td>
<td>taken</td>
<td>strongly taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>strongly taken</td>
<td>taken</td>
<td>strongly taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>strongly taken</td>
<td>taken</td>
<td>strongly taken</td>
</tr>
</tbody>
</table>
Two-bit Prediction

• The two bit prediction scheme is used very widely and in many ways.
  • Make a table of 2-bit predictors
  • Devise a way to associate a 2-bit predictor with each dynamic branch
  • Use the 2-bit predictor for each branch to make the prediction.

• In the previous example we associated the predictors with branches using the PC.
  • We’ll call this “per-PC” prediction.

Per-PC Predictor

Table of $2^n$ 2-bit predictors

n low-order bits of the PC → Prediction

Branch outcome
Associating 2-bit Predictors with Branches: Using the low-order PC bits

- When is branch behavior predictable?
  - Loops -- for(i = 0; i < 10; i++) {} 9 taken branches, 1 not-taken branch. All 10 are pretty predictable.
  - Run-time constants
    - Foo(int v,) { for (i = 0; i < 1000; i++) {if (v) {...}}}.  
    - The branch is always taken or not taken.
  - Corollated control
    - a = 10; b = <something usually larger than a >
    - if (a > 10) {}
    - if (b > 10) {}
  - Function calls
    - LibraryFunction() -- Converts to a jr (jump register) instruction, but it’s always the same.
    - BaseClass * t; // t is usually a of sub class, SubClass
    - t->SomeVirtualFunction() // will usually call the same function

OK -- we miss one per loop
Good
Poor -- no help
Not applicable
Predicting Loop Branches Revisited

while (1) {
    for(j = 0; j < 3; j++) {
    }
}

- What’s the pattern we need to identify?
Predicting Loop Branches Revisited

while (1) {
  for (j = 0; j < 3; j++) {
  }
}

<table>
<thead>
<tr>
<th>iteration</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
</tr>
<tr>
<td>1</td>
<td>taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
</tr>
<tr>
<td>1</td>
<td>taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
</tr>
</tbody>
</table>

• What’s the pattern we need to identify?
Dynamic prediction 4: Global branch history

• Instead of using the PC to choose the predictor, use a bit vector made up of the previous branch outcomes.
Dynamic prediction 4: Global branch history

- Instead of using the PC to choose the predictor, use a bit vector made up of the previous branch outcomes.

<table>
<thead>
<tr>
<th>iteration</th>
<th>Actual</th>
<th>Branch history</th>
<th>Steady state prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>taken</td>
<td>11111</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>11111</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>11111</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
<td>11111</td>
<td></td>
</tr>
<tr>
<td>outer loop branch</td>
<td>taken</td>
<td>11110</td>
<td>taken</td>
</tr>
<tr>
<td>1</td>
<td>taken</td>
<td>11011</td>
<td>taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>11011</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>10111</td>
<td>taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
<td>01111</td>
<td>not taken</td>
</tr>
<tr>
<td>outer loop branch</td>
<td>taken</td>
<td>11110</td>
<td>taken</td>
</tr>
<tr>
<td>1</td>
<td>taken</td>
<td>11101</td>
<td>taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>11011</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>10111</td>
<td>taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
<td>01111</td>
<td>not taken</td>
</tr>
<tr>
<td>outer loop branch</td>
<td>taken</td>
<td>11110</td>
<td>taken</td>
</tr>
<tr>
<td>1</td>
<td>taken</td>
<td>11101</td>
<td>taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>11011</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>10111</td>
<td>taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
<td>01111</td>
<td>not taken</td>
</tr>
<tr>
<td>outer loop branch</td>
<td>taken</td>
<td>11110</td>
<td>taken</td>
</tr>
</tbody>
</table>
Dynamic prediction 4: Global branch history

- Instead of using the PC to choose the predictor, use a bit vector made up of the previous branch outcomes.

<table>
<thead>
<tr>
<th>iteration</th>
<th>Actual</th>
<th>Branch history</th>
<th>Steady state prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>taken</td>
<td>11111</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>11111</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>11111</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
<td>11111</td>
<td></td>
</tr>
<tr>
<td>outer loop branch</td>
<td>taken</td>
<td>11110</td>
<td>taken</td>
</tr>
<tr>
<td>1</td>
<td>taken</td>
<td>11101</td>
<td>taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>11011</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>10111</td>
<td>taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
<td>01111</td>
<td>not taken</td>
</tr>
<tr>
<td>outer loop branch</td>
<td>taken</td>
<td>11110</td>
<td>taken</td>
</tr>
<tr>
<td>1</td>
<td>taken</td>
<td>11101</td>
<td>taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>11011</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>10111</td>
<td>taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
<td>01111</td>
<td>not taken</td>
</tr>
<tr>
<td>outer loop branch</td>
<td>taken</td>
<td>11110</td>
<td>taken</td>
</tr>
<tr>
<td>1</td>
<td>taken</td>
<td>11101</td>
<td>taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>11011</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>10111</td>
<td>taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
<td>01111</td>
<td>not taken</td>
</tr>
</tbody>
</table>

Nearly perfect
Dynamic prediction 4: Global branch history

- Instead of using the PC to choose the predictor, use a bit vector made up of the previous branch outcomes.
Dynamic prediction 4: Global branch history

• How long should the history be?

• Imagine N bits of history and a loop that executes K iterations
  • If $K \leq N$, history will do well.
  • If $K > N$, history will do poorly, since the history register will always be all 1’s for the last $K-N$ iterations. We will mis-predict the last branch.
Dynamic prediction 4: Global branch history

• How long should the history be?
  
  Infinite is a bad choice. We would learn nothing.

• Imagine N bits of history and a loop that executes K iterations
  
  • If $K \leq N$, history will do well.
  • If $K > N$, history will do poorly, since the history register will always be all 1’s for the last $K-N$ iterations. We will mis-predict the last branch.
Associating Predictors with Branches: 
Global history

• When is branch behavior predictable?
  • Loops -- for(i = 0; i < 10; i++) {} 9 taken branches, 
    1 not-taken branch. All 10 are pretty predictable.
  • Run-time constants
    • Foo(int v,) { for (i = 0; i < 1000; i++) {if (v) {...}}}.
    • The branch is always taken or not taken.
  • Corollated control
    • a = 10; b = <something usually larger than a>
    • if (a > 10) {}
    • if (b > 10) {}
  • Function calls
    • LibraryFunction() -- Converts to a jr (jump register) 
      instruction, but it’s always the same.
    • BaseClass * t; // t is usually a of sub class, SubClass
    • t->SomeVirtualFunction() // will usually call the same function

Good

Not so great

Pretty good, as long as 
the history is not too long

Not

applicable
The Local History Predictor

- Use a table of history registers, indexed by the low-order bits of the PC.
- Also use the PC to choose a table, each indexed by the history for that branch.
- For loops this does better than global history.
  - `Foo() { for(i = 0; i < 4; i++){} }`.
  - If `foo` is called from many places, the global history will be polluted, but the local history for the loop’s branch will be kept safe.
Other Ways of Identifying Branches

- Combine Global History and bits of the PC
  - Gshare predictor
  - Index a of two-bit predictors with the PC XOR Global History.

![GShare Predictor Diagram]

- n bits of the global history
- XOR
- n low-order bits of the PC
- Branch outcome
- Table of $2^n$ 2-bit predictors
- Prediction
Other Ways of Identifying Branches

- How do we get the best of all possible worlds?
- Build them all, and have a predictor to decide which one to use on a given branch -- The Hybrid (or Tournament) Predictor
  - 2-bit predictor now has different states
  - Strongly prefer GShare, weakly prefer Gshare, weakly prefer local, strongly prefer local.
Other Ways of Identifying Branches

- How do we get the best of all possible worlds?
- Build them all, and have a predictor to decide which one to use on a given branch -- The Hybrid (or Tournament) Predictor
  - 2-bit predictor now has different states
  - Strongly prefer GShare, weakly prefer Gshare, weakly prefer local, strongly prefer local.
The Hybrid Predictor

• Loops -- for(i = 0; i < 10; i++) {}  9 taken branches, 1 not-taken branch. All 10 are pretty predictable.  Good

• Run-time constants
  • Foo(int v,) { for (i = 0; i < 1000; i++) {if (v) {...}}}.  Good
  • The branch is always taken or not taken.  Good

• Corollated control
  • a = 10;  b = <something usually larger than a >  Good
  • if (a > 10) {}
  • if (b > 10) {}

• Function invocations
  • LibraryFunction() -- Converts to a jr (jump register) instruction, but it’s always the same.  Not applicable
  • BaseClass * t;  // t is usually a of sub class, SubClass
  • t->SomeVirtualFunction() // will usually call the same function

• Function Returns
  • You have to jump back to where you came from after a function call.  Not applicable
Interference

• Our schemes for associating branches with predictors are imperfect.
• Different branches may map to the same predictor and pollute the predictor.
• This is called “destructive interference”
• Using larger tables will (typically) reduce this effect.
Predicting Function Invocations

- **Branch Target Buffers (BTB)**
  - Use a table, indexed by PC, that stores the last target of the jump.
  - When you fetch a jump, start executing at the address in the BTB.
  - Update the BTB when you find out the correct destination.

- The BTB is useful for predicting function calls and jump instructions (and some other things, as we will see shortly.)
Predicting Returns

- Function call returns are hard to predict
  - For every call site, the return address is different
  - The BTB will do a poor job, since it’s based on PC
- Instead, maintain a “return stack predictor”
  - Keep a stack of return targets
  - `jal` pushes $ra onto the stack
  - Fetch predicts the target for `ret`n instruction by popping an address off the stack.
  - Doesn’t work in MIPS, because there is no return instruction.

Return Address Predictor

- Push on `jal`
- Pop on `ret`n
- Stack of return addresses
Predicting Returns In MIPS

- The return address predictor doesn’t work in MIPS, because there is no return instruction
- How could we fix it?
The return address predictor doesn't work in MIPS, because there is no return instruction.

How could we fix it?

- Add a retn instruction -- it's just jr but with a different opcode so we can tell the difference.
- Build a predictor to choose between the return address predictor and the BTB.
Modern processors have deep pipelines.

Conditional branch predictors are good, but they can take several cycles to make a prediction.

What do we fetch in the meantime?

Processors will predict each branch multiple times.

First, use the BTB -- The accuracy may not be great.

A few cycles later, the conditional branch predictor lets you know if the BTB was probably right or wrong.

Several cycles after that, the actual branch direction is known.