Last Time: Error Correcting Codes

We write: \[ D_3D_2D_1P_2D_0P_1P_0 \]
0 1 1 0 0 1 1

Later, we read: \[ D_3D_2D_1P_2D_0P_1P_0 \]
0 1 0 0 0 1 1

Cosmic ray hit D1. But how do we know that?

On readout we compute:

\[ P_0 \text{xor } D_3 \text{xor } D_1 \text{xor } D_0 = 1 \text{xor } 0 \text{xor } 0 \text{xor } 0 = 1 \]
\[ P_1 \text{xor } D_3 \text{xor } D_2 \text{xor } D_0 = 1 \text{xor } 0 \text{xor } 1 \text{xor } 0 = 0 \]
\[ P_2 \text{xor } D_3 \text{xor } D_2 \text{xor } D_1 = 0 \text{xor } 0 \text{xor } 1 \text{xor } 0 = 1 \]

What does “5” mean?

Note: we number the least significant bit with 1, not 0!
0 is reserved for “no errors”.

The position of the flipped bit!
To repair, just flip it back ...

7 6 5 4 3 2 1
\[ D_3D_2D_1P_2D_0P_1P_0 \]
0 1 0 0 0 1 1
Today: Beyond the 5-stage pipeline

* Taxonomy: Introduction to advanced processor techniques.

* Superpipelining: Increasing the number of pipeline stages.

* Superscalar: Issuing several instructions in a single cycle.
At best, the 5-stage pipeline executes one instruction per clock, with a clock period determined by the slowest stage.

Application does not need multi-cycle instructions (multiply, divide, etc.)
Superpipelining: Add more stages

Goal: Reduce critical path by adding more pipeline stages.

Example: 8-stage ARM XScale: extra IF, ID, data cache stages.

Difficulties: Added penalties for load delays and branch misses.

Ultimate Limiter: As logic delay goes to 0, FF clk-to-Q and setup.
Superscalar: Multiple issues per cycle 

Goal: Improve CPI by issuing several instructions per cycle.

Example: CPU with floating point ALUs: issue 1 FP + 1 integer instruction per cycle.

Difficulties: Load and branch delays affect more instructions.

Ultimate Limiter: Programs may be a poor match to issue rules.
Out of Order: Going around stalls

Goal: Issue instructions out of program order

Example:

... so let ADDD go first

Difficulties: Bookkeeping is highly complex. A poor fit for lockstep instruction scheduling.

Ultimate Limiter: The amount of instruction level parallelism present in an application.
Dynamic Scheduling: End lockstep

Goal: Enable out-of-order by breaking pipeline in two: fetch and execution.

Example: IBM Power 5:

Limiters: Design complexity, instruction level parallelism.
Throughput and multiple threads

Goal: Use multiple CPUs (real and virtual) to improve (1) throughput of machines that run many programs (2) execution time of multithreaded programs.

Example: Sun Niagara (8 SPARCs on one chip).

Difficulties: Gaining full advantage requires rewriting applications, OS, libraries.

Ultimate limiter: Amdahl’s law, memory system performance.
Reminder: Friday Test Bench Checkoff

Test vector suite for Week 3/4/5 checkoffs, running in ModelSim (3/4) and SPIM (5).

Detailed block diagrams, state machines, and Lab 3 CPU changes
Superpipelining
Add pipeline stages, reduce clock period

Q. Could adding pipeline stages reduce CPI for an application?
A. Yes, due to these problems:

<table>
<thead>
<tr>
<th>CPI Problem</th>
<th>Possible Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Taken branches cause longer stalls</td>
<td>Branch prediction, loop unrolling</td>
</tr>
<tr>
<td>Cache misses take more clock cycles</td>
<td>Larger caches, add prefetch opcodes to ISA</td>
</tr>
</tbody>
</table>
Recall: Control hazards ...

We avoiding stalling by (1) adding a branch delay slot, and (2) adding comparator to ID stage
If we add more early stages, we must stall.

Sample Program
(ISA w/o branch delay slot)

I1: BEQ R4,R3,25
I2: AND R6,R5,R4
I3: SUB R1,R9,R8

<table>
<thead>
<tr>
<th>Time</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
<th>t7</th>
<th>t8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I1</td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td>EX</td>
<td></td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>I2</td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I3</td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

EX stage computes if branch is taken
If branch is taken, these instructions MUST NOT complete!
Solution: Branch prediction ...

We update the PC based on the outputs of the branch predictor. If it is perfect, pipe stays full!

Dynamic Predictors: a cache of branch history

Time: t1  t2  t3  t4  t5  t6  t7  t8
Inst
I1:  IF  ID  EX
I2:  IF  ID
I3:  IF
I4:  
I5:  
I6:  

EX stage computes if branch is taken

If we predicted incorrectly, these instructions MUST NOT complete!

A control instr?

Taken or Not Taken?

The PC a branch "targets"
Branch predictors cache branch history

Address of BNEZ instruction
0b0110[...]01001000

Branch Target Buffer (BTB)
28-bit address tag | target address

0b0110[...]0100 | PC + 4 + Loop

BNEZ R1 Loop

Branch History Table (BHT)

Update
BHT/BTB
for next
time, once
ture
behavior
known

"Taken" Address
Must check prediction, kill instructions if needed.

"Taken" or "Not Taken"
Simple ("2-bit") Branch History Table Entry

Prediction for next branch
(1 = take, 0 = not take)

Was last prediction correct?
(1 = yes, 0 = no)

We do not change the prediction the first time it is incorrect. Why?

```
ADDI R4, R0, 11
loop:   SUBI R4, R4, -1
       BNE R4, R0, loop
```

This branch taken 10 times, then not taken once (end of loop). The next time we enter the loop, we would like to predict “take” the first time through.
Spatial enhancements: many BHTs...

```
0b0110[...]01001000  BNEZ  R1  Loop
```

Branch History Tables

- (BHT00)
- (BHT01)
- (BHT10)
- (BHT11)

Detects patterns in:
- if (x < 12) [...]
- if (x < 6) [...]

code.

Update the table whose value was used for the branch instruction.

Yeh and Patt, 1992.

Were last two branches in instruction stream “taken” or not?

“Taken” or “Not Taken”
Hardware limits to superpipelining?

**FO4 Delays**

**CPU Clock Periods 1985-2005**

- **MIPS 2000** 5 stages
- **Pentium Pro** 10 stages
- **Pentium 4** 20 stages

**Cell: 11 FO4 delays**

**FO4: How many fanout-of-4 inverter delays in the clock period.**

Thanks to Francois Labonte, Stanford
Superscalar
Example: Superscalar MIPS. Fetches 2 instructions at a time. If first integer and second floating point, issue in same cycle

<table>
<thead>
<tr>
<th>Integer instruction</th>
<th>FP instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F0,0(R1)</td>
<td>ADDD F4,F0,F2</td>
</tr>
<tr>
<td>LD F6,-8(R1)</td>
<td>ADDD F8,F6,F2</td>
</tr>
<tr>
<td>LD F10,-16(R1)</td>
<td>ADDD F12,F10,F2</td>
</tr>
<tr>
<td>LD F14,-24(R1)</td>
<td>ADDD F16,F14,F2</td>
</tr>
<tr>
<td>LD F18,-32(R1)</td>
<td>ADDD F20,F18,F2</td>
</tr>
<tr>
<td>SD 0(R1),F4</td>
<td>ADDD F4,F0,F2</td>
</tr>
<tr>
<td>SD -8(R1),F8</td>
<td>ADDD F8,F6,F2</td>
</tr>
<tr>
<td>SD -16(R1),F12</td>
<td>ADDD F12,F10,F2</td>
</tr>
<tr>
<td>SD -24(R1),F16</td>
<td>ADDD F16,F14,F2</td>
</tr>
</tbody>
</table>

Two issues per cycle

One issue per cycle
Superscalar: Visualizing the pipeline

Three instructions affected by a single cycle of load delay. Why?
Limitations of “lockstep” superscalar

* Only get 0.5 CPI for a 50/50 float/int mix with no hazards. For games/media, this may be OK.

* Extending scheme to speed up general apps (Microsoft Office, ...) is complicated.

* If one accepts building a complicated machine, there are better ways to do it.

Next time: Dynamic Scheduling
Conclusion: Superpipelining, Superscalar

The 5 stage pipeline: a starting point for performance enhancements, a building block for multiprocessing.

Superpipelining: Reduce critical path by adding more pipeline stages. Has the potential to increase the CPI.

Superscalar: Multiple instructions at once. Programs must fit the issue rules. Adds complexity.