CS 152
Computer Architecture and Engineering

Lecture 17 – Advanced Processors I

2006-10-26

John Lazzaro
(www.cs.berkeley.edu/~lazzaro)

TAs: Udam Saini and Jue Sun

www-inst.eecs.berkeley.edu/~cs152/
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:

\[ P_2 \text{xor} D_3 \text{xor} D_2 \text{xor} D_1 = 0 \text{xor} 0 \text{xor} 1 \text{xor} 0 = 1 = C_2 \]
\[ P_1 \text{xor} D_3 \text{xor} D_2 \text{xor} D_0 = 1 \text{xor} 0 \text{xor} 1 \text{xor} 0 = 0 = C_1 \]
\[ P_0 \text{xor} D_3 \text{xor} D_1 \text{xor} D_0 = 1 \text{xor} 0 \text{xor} 0 \text{xor} 0 = 1 = C_0 \]

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.

Processor has no “multi-cycle” instructions (ex: multiply with an accumulate register)
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.

Also, power!
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

<table>
<thead>
<tr>
<th>Seconds Program</th>
<th>Instructions Program</th>
<th>Cycles Instruction</th>
<th>Seconds Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>F2, 34(R2)</td>
<td>latency 1</td>
<td></td>
</tr>
<tr>
<td>LD</td>
<td>F4, 45(R3)</td>
<td>long</td>
<td></td>
</tr>
<tr>
<td>MULTD</td>
<td>F6, F4, F2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>ADDD</td>
<td>F8, F2, F2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

### Goal: Issue instructions out of program order

#### Example:

... so let **ADDD** go first

**MULTD** waiting on F4 to load ...

### 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 multi-threaded programs.

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

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

Ultimate limiter: Amdahl’s law, memory system performance.
Superpipelining
Note: Some stages now overlap, some instructions take extra stages.

IF now takes 2 stages (pipelined I-cache)

ID and RF each get a stage.

ALU split over 3 stages

MEM takes 2 stages (pipelined D-cache)
Superpipelining techniques ...

- Split **ALU** and **decode** logic over several pipeline stages.

- **Pipeline memory**: Use more banks of smaller arrays, add pipeline stages between decoders, muxes.

- Remove “rarely-used” **forwarding networks** that are on critical path.
  - Creates stalls, affects CPI.

- **Pipeline** the wires of frequently used **forwarding networks**.

*Also: Clocking tricks (example: negedge register file in book pipeline)*
Architects specify number of rows and columns. Word and bit lines slow down as array grows larger!

Parallel Data I/O Lines

Add muxes to select subset of bits

How could we pipeline this memory?
ALU: Pipelining Unsigned Multiply

Facts to remember

- Multiplicand: 1101 (13)
- Multiplier: 1011 (11)
- Partial products:
  - 1101
  - 1101
  - 0000
  - 1101
- Product: 10001111 (143)

m bits x n bits = m+n bit product

Binary makes it easy:
- 0 => place 0 (0 x multiplicand)
- 1 => place a copy (1 x multiplicand)
Building Block: Full-Adder Variant

1-bit signals: $x, y, z, s, \text{Cin}, \text{Cout}$

- $z$: one bit of multiplier
- $x$: one bit of multiplicand
- $y$: one bit of the "running sum"

If $z = 1$, \( \{\text{Cout}, s\} \leq x + y + \text{Cin} \)
If $z = 0$, \( \{\text{Cout}, s\} \leq y + \text{Cin} \)
Put it together: Array computes \( P = A \times B \)

To pipeline array:

Place registers between adder stages.

Use registers to delay selected \( A \) and \( B \) bits.

As drawn, combinational (slow!).
In Xilinx: Compose built-in multipliers

Note: All inputs to the multiplier or adder less than required are sign extended.

Indicates optional pipeline stages.
Recall: IBM Power Timing Closure

“Pipeline engineering” happens here ...

Add pipeline stages, reduce clock period

Q. Could adding pipeline stages hurt the 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

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

EX stage computes if branch is taken

If branch is taken, these instructions MUST NOT complete!
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

If we predicted incorrectly, these instructions MUST NOT complete!
Branch predictors cache branch history

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

Branch Target Buffer (BTB)

28-bit address tag

0b0110[...]0100

PC + 4 + Loop

BNEZ R1 Loop

Branch History Table (BHT)

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

"Taken" or "Not Taken"

"Taken" Address

Hit

Must check prediction, kill instruction if needed.
Simple ("2-bit") Branch History Table Entry

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

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

D Q

After we “check” prediction ...

Flip bit if prediction is not correct and “last predict correct” bit is 0.

Set to 1 if prediction bit was correct.
Set to 0 if prediction bit was incorrect.
Set to 1 if prediction bit flips.

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

BHT00/01/10/11
code the last four branches in the instruction stream

"Taken" or "Not Taken"

Branch History Tables

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

Adaptive function of history, state

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

Yeh and Patt, 1992.
Hardware limits to superpipelining?

MIPS 2000
5 stages

Pentium Pro
10 stages

Pentium 4
20 stages

Historical limit: about 12

Power wall: Intel Core Duo has 14 stages

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

Thanks to Francois Labonte, Stanford
Superscalar

Basic Idea: Improve CPI by issuing several instructions per cycle.
Superscalar R machine

Instruction Issue Logic

Data
Instr
Mem

PC and Sequencer

RegFile
rs1
rs2
ws1
wd1
rs3
rs4
ws2
wd2

A
B

WE1
WE2

R

Y

ALU

op
32
32
32
32

64
32
32
32

IR

IR

IR

IR

IR

IR

IR

IF (Fetch)

ID (Decode)

EX (ALU)

MEM

WB

CS 152 L17: Advanced Processors I

UC Regents Fall 2006 © UCB
Sustaining Dual Instr Issues (no forwarding)

ADD R8, R0, R0
ADD R11, R0, R0
ADD R27, R26, R25
ADD R30, R29, R28
ADD R21, R20, R19
ADD R24, R23, R22
ADD R15, R14, R13
ADD R18, R17, R16
ADD R9, R8, R7
ADD R12, R11, R10

It's rarely this good...
We add 12 forwarding buses (not shown). (6 to each ID from stages of both pipes).

Worst-Case Instruction Issue
ADD R8, R0, R0
ADD R9, R8, R0
ADD R10, R9, R0
ADD R11, R10, R0

Dependencies force “serialization”
Superscalar: A simple example ...

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></td>
</tr>
<tr>
<td>SD -8(R1),F8</td>
<td></td>
</tr>
<tr>
<td>SD -16(R1),F12</td>
<td></td>
</tr>
<tr>
<td>SD -24(R1),F16</td>
<td></td>
</tr>
</tbody>
</table>

Why is the control for this CPU not so hard to do?

Two issues per cycle

One issue per cycle
Three instructions potentially affected by a single cycle of load delay, as FP register loads done in the “integer” pipeline.
Limitations of “lockstep” superscalar

* Gets 0.5 CPI only 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 hurt the CPI.

Superscalar: Multiple instructions at once. Programs must fit the issue rules. Adds complexity.
Run your test vector suite on the Calinx board, display results on LEDs.

Write different addresses, values!

Test ongoing transactions on both buses, with randomized start times. Load, store, and verify different data word patterns.

Today 1PM: Mid-Term Regrade Deadline