Today: First advanced processor lecture

Super-pipelining: Beyond 5 stages.

Short Break.

Branch prediction: Can we escape control hazards in long CPU pipelines?
From Appendix C: Filling the branch delay slot

The top box in each pair shows the code before scheduling; the bottom box shows the scheduled code. In (a), the delay slot is scheduled with an independent instruction from before the branch. This is the best choice. Strategies (b) and (c) are used when (a) is not possible. In the code sequences for (b) and (c), the use of R1 in the branch condition prevents the DADD instruction (whose destination is R1) from being moved after the branch. In (b), the branch delay slot is scheduled from the target of the branch; usually the target instruction will need to be copied because it can be reached by another path. Strategy (b) is preferred when the branch is taken with high probability, such as a loop branch. Finally, the branch may be scheduled from the not-taken fall-through as in (c). To make this optimization legal for (b) or (c), it must be OK to execute the moved instruction when the branch goes in the unexpected direction. By OK we mean that the work is wasted, but the program will still execute correctly. This is the case, for example, in (c) if R7 were an unused temporary register when the branch goes in the unexpected direction.
Superpipelining
5 Stage Pipeline: A point of departure

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!
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: use positive-edge AND negative-edge flip-flops)
Recall: IBM Power Timing Closure

Pipelining a 256 byte instruction memory.

Fully combinational (and slow). Only read behavior shown.

A7-A0: 8-bit read address

Can we add two pipeline stages?

Each register holds 32 bytes (256 bits)

OE --> Tri-state Q outputs!

Data output is 32 bits, i.e. 4 bytes

CS 152: L6: Superpipelining + Branch Prediction
On a chip: “Registers” become SRAM cells

Architects specify number of rows and columns. Word and bit lines slow down as array grows larger!

--Diagram of SRAM array--

Parallel Data I/O Lines

How could we pipeline this memory? See last slide.
5.85 million devices

0.65 million devices

RISC CPU

D-CACHE

I-CACHE

CLOCK SPINE

BUS

EXE

MAC

CONTROL

DECODE

JTAG
IC processes are optimized for small SRAM cells

From Marvell ARM CPU paper: 90% of the 6.5 million transistors, and 60% of the chip area, is devoted to cache memories.

Implication? SRAM is 6X as dense as logic.
RAM Compilers

On average, 30% of a modern logic chip is SRAM, which is generated by RAM compilers.

Compile-time parameters set number of bits, aspect ratio, ports, etc.
**Facts to remember**

- 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

If \(z = 1\), \(\{\text{Cout}, s\} \leq x + y + \text{Cin}\)

If \(z = 0\), \(\{\text{Cout}, s\} \leq y + \text{Cin}\)

\(x\): one bit of multiplicand

\(y\): one bit of the “running sum”
Put it together: Array computes \( P = A \times B \)

**To pipeline array:**

Place registers between adder stages (in green).

Add registers to delay selected \( A \) and \( B \) bits (not shown).

**Fully combinational implementation is slow!**
Adding pipeline stages is not enough ...

MIPS R4000: Simple 8-stage pipeline

Branch stalls are the main reason why pipeline CPI > 1.

2-cycle load delay, 3-cycle branch delay.

(Appendix C, Figure C.52)
Branch Prediction
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
I2: IF ID
I3: IF
I4: 
I5: 
I6: 

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

Branch Predictor

Predictions

A control instr?

Taken or Not? where to?

Taken? What PC?

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

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

Branch instruction
BNEZ R1 Loop

Branch Target Buffer (BTB)

30-bit address tag

target address

4096 entries ...

At EX stage, update BTB/BHT, kill instructions, if necessary.

"Hit"

"Taken" Address

2 state bits

"Taken" or "Not Taken"

Branch History Table (BHT)

Drawn as fully associative to focus on the essentials.

In real designs, always direct-mapped.

"Taken" or "Not Taken"
Branch predictor: direct-mapped version

Address of BNEZ instruction
0b011[..]010[..]100

Branch Target Buffer (BTB)
- 18-bit address tag
- target address

Branch History Table (BHT)
- 12 bits
- 4096 BTB/BHT entries

BNEZ R1 Loop

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

"Taken" or "Not Taken"
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 ...**

- **D**
- **Q**

**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?**

```assembly
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.
The misprediction rate for the integer benchmarks (gcc, espresso, eqntott, and li) is substantially higher (average of 11%) than that for the floating-point programs (average of 4%). Omitting the floating-point kernels (nasa7, matrix300, and tomcatv) still yields a higher accuracy for the FP benchmarks than for the integer benchmarks. These data, as well as the rest of the data in this section, are taken from a branch-prediction study done using the IBM Power architecture and optimized code for that system. See Pan, So, and Rameh [1992]. Although these data are for an older version of a subset of the SPEC benchmarks, the newer benchmarks are larger and would show slightly worse behavior, especially for the integer benchmarks.

**Figure C.19**

- NASA7: 1%
- Matrix300: 0%
- Tomcatv: 1%
- Doduc: 5%
- Spice: 9%
- Fppp: 9%
- Gcc: 12%
- Espresso: 5%
- Eqntott: 18%
- Li: 10%
Branch Prediction: Trust, but verify ...

Instr Fetch → Decode & Reg Fetch → Execute

PC

I-Cache

Addr Data

IR

RegFile

rd1

rd2

WE

wd

rs1

rs2

ws

Predicted PC

Branch Predictor and BTB

Predictions

A branch instr?

Taken or Not Taken?

If taken, where to? What PC?

Branch Predictor

and BTB

Predictions

Note instruction type and branch target. Pass to next stage.

Prediction info -->

Branch

Taken/Not Taken

Check all predictions. Take actions if needed (kill instructions, update predictor).

Prediction info -->

Logic

If taken, where to? What PC?

Branch

Taken/Not Taken

Check all predictions. Take actions if needed (kill instructions, update predictor).

Prediction info -->

Logic

If taken, where to? What PC?

Branch

Taken/Not Taken

Check all predictions. Take actions if needed (kill instructions, update predictor).

Prediction info -->

Logic

If taken, where to? What PC?

Branch

Taken/Not Taken

Check all predictions. Take actions if needed (kill instructions, update predictor).

Prediction info -->

Logic

If taken, where to? What PC?

Branch

Taken/Not Taken

Check all predictions. Take actions if needed (kill instructions, update predictor).

Prediction info -->

Logic

If taken, where to? What PC?

Branch

Taken/Not Taken

Check all predictions. Take actions if needed (kill instructions, update predictor).

Prediction info -->

Logic

If taken, where to? What PC?

Branch

Taken/Not Taken

Check all predictions. Take actions if needed (kill instructions, update predictor).

Prediction info -->

Logic

If taken, where to? What PC?

Branch

Taken/Not Taken

Check all predictions. Take actions if needed (kill instructions, update predictor).

Prediction info -->

Logic

If taken, where to? What PC?

Branch

Taken/Not Taken

Check all predictions. Take actions if needed (kill instructions, update predictor).

Prediction info -->

Logic

If taken, where to? What PC?
Flowchart control for dynamic branch prediction.

Figure 3.22 The steps involved in handling an instruction with a branch-target buffer.
Spatial Predictors

C code snippet:

```c
b1 if (aa==2) 
  aa=0;

b2 if (bb==2) 
  bb=0;

b3 if (aa!=bb) {
```

Idea: Devote hardware to four 2-bit predictors for BEQZ branch.

- **P1**: Use if b1 and b2 not taken.
- **P2**: Use if b1 taken, b2 not taken.
- **P3**: Use if b1 not taken, b2 taken.
- **P4**: Use if b1 and b2 taken.

Track the current taken/not-taken status of b1 and b2, and use it to choose from P1 ... P4 for BEQZ.

After compilation:

```
L1: DADDIU R3,R1,#-2
    BNEZ R3,L1
    DADD R1,R0,R0
    DADDIU R3,R2,#-2
    BNEZ R3,L2
    DADD R2,R0,R0
    DSUBU R3,R1,R2
    BEQZ R3,L3
```

Can b1 and b2 help us predict it?

We want to predict this branch.
Branch History Register: Tracks global history

**Intr Fetc**h | **Decode & Reg Fetch** | We choose which predictor to use (and update) based on the Branch History Register.

**Branch Predictor and BTB**

A branch instr? Taken or Not Taken? If taken, where to? What PC?

**Logic**

Shift register. Holds taken/not-taken status of last 2 branches.
Spatial branch predictor (BTB, tag not shown)

```
0b0110[...]01001000  BEQZ  R3  L3
```

Branch History Tables

- **P1**: 2 state bits
- **P2**: 2 state bits
- **P3**: 2 state bits
- **P4**: 2 state bits

**Map PC to index**

**Branch History Register**

- **D Q WE △**
  - (bb==2) branch
  - (aa==2) branch

**Mux to choose “which branch predictor”**

- “Taken” or “Not Taken”
- For (aa!= bb) branch

**Detects patterns in:**

- if (aa==2) aa=0;
- if (bb==2) bb=0;
- if (aa!=bb) { code.

- Yeh and Patt, 1992.

- 95% accurate
For more details on branch prediction:

Appendix C.2.6, Chapter 3.3.9
Spatial Branch Predictor [paper]

Figure 3.3
Predict function returns by stacking call info

Figure 3.24
Hardware limits to superpipelining?

FO4 Delays

Historical limit: about 12 FO4s

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

Thanks to Francois Labonte, Stanford

CPU Clock Periods 1985-2005

MIPS 2000
5 stages

Pentium Pro
10 stages

Pentium 4
20 stages

Power wall:
Intel Core Duo has 14 stages

Thanks to Francois Labonte, Stanford
FO4 Delays Per Cycle for Processor Designs

 FO4 delay per cycle is roughly proportional to the amount of computation completed per cycle.
On Tuesday

We turn our focus to memory system design ...

<table>
<thead>
<tr>
<th>T</th>
<th>Memory</th>
<th>Chapter 2.3.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2/11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Have a good weekend!