Chapter 4

The Processor
Control & Datapath

Single (long) cycle per instruction
Pipelined Datapath & Control Operation without data or control dependencies, yet

University of Crete
Dept. of Computer Science
CS–225 (HY–225)
Computer ORganization
Spring 2020 semester

Slides for §9.3 – 9.5

§9.3 Pipelined Datapath Operation
§9.4 Control for the Pipelined Datapath
§9.5 Graphical representation: time–work
60: \textit{ld} \ x_{10}, \ 40(x_{1})
64: \textit{sub} \ x_{11}, \ x_{2}, \ x_{3}
68: \textit{add} \ x_{12}, \ x_{3}, \ x_{4}
72: \textit{ld} \ x_{13}, \ 48(x_{1})
76: \textit{add} \ x_{14}, \ x_{5}, \ x_{6}

\textbf{Cycle 5}

\begin{itemize}
  \item Instr. Fetch
  \item Reg.Rd; Op
  \item ALU
  \item Data Mem.
  \item Write Back
\end{itemize}
Cycle 5

60: `ld x10, 40(x1)`
64: `sub x11, x2, x3`
68: `add x12, x3, x4`
72: `sd x13, 48(x1)`
76: `add x14, x5, x6`
Data Dependences (Hazards) in Pipelines

- RAW (Read after Write) – true dependence, as above
- RAR (Read after Read) – not a dependence, can freely reorder the reads
- WAR (Write after Read) – "antidependence": if you want to do the write (I2) early, just keep a copy of the old data and have I1 read that copy
- WAW (Write after Write) – if you want to reorder them, simply abort the write of I1 (if no one reads this word between I1 and I2)

I2 needs the new data written by I1, hence must wait for I1 to write – or at least to generate – the new data.
No Memory Data Hazards in our simple Pipeline

- Data Memory accesses are performed ‘in–order’ in our simple pipeline, i.e. are not reordered relative to what the program specifies, thus, no dependences of memory word accesses are ever violated.
For each instruction that writes a destination register, if the next 2 or 3 instructions read that same register, i.e. need its result, we have to do something about it...
Actual Need–Produce Time vs. from/in–Register Time

- All we care about is actual Results ‘Forwarded’ from Producer to Consumer instruction
- We can ‘Bypass’ the ‘official’ loop through the Register File for immediate–use Results
ALU result to next I; Load result to next–after–next I

from **ALU** Instructions:

- ALU op. result can be *forwarded* to any subsequent instruction

from **Load** Instruction:

- Loaded data can NOT be used by the one immediately succeeding instruction, without it having to wait one extra cycle

- ALU instructions never stall the pipeline, but Load instructions will do so when immediately followed by a dependent instruction
Distance 1 dependence on LOAD: Wait!

- The instruction immediately after a Load wants to use the load’ed data
- Impossible without losing one cycle:
  - force this instruction to wait (repeat itself on the next cycle)
- Simple, in−order Pipeline: the next instruction has to wait too!
If unknown to compiler, static sch. impossible => dynamic scheduling at runtime (ooo pipe)

Does the compiler know for sure if i!=j

e = b − f;
a = b + c;

What if the program is?: RAW dependence?

the more things you have 'up in the air' (in parallel), the more temporary registers you need in order to 'name' those 'pending' values

This is 'Static' Scheduling, at Compile Time

- Does the compiler know for sure if i!=j (OK to reorder sd−ld) or i==j (fwd in reg.)?
- If unknown to compiler, static sch. impossible => dynamic scheduling at runtime (ooo pipe)
Control Dependences (branch/jump) in Pipelines

- ‘Data Dependence’ = next instruction uses data (register/memory) from previous
- ‘Control Dependence’ = which is the next instruction depends on the previous
- Control Dependences arise from ‘Control Transfer Instructions (CTI)’
- Control Transfer Instructions (CTI) are: Jump and Branch Instructions
- ‘Jumps’ are Unconditional CTI’s: they always transfer control
- ‘Branches’ are Conditional CTI’s: whether or not they transfer control depends on the result of a data comparison that they have to perform

Statistics (rough numbers, in a majority of programs, but NOT always so):

- Branches are about 15–16% of all (‘dynamically’) executed instructions in a program
  - about 2/3 of executed branches are ‘taken’ (successful) = ~10% of all instr.
  - about 1/3 of executed branches are not taken (unsuccessful) = ~5% of all instr.
  - most backwards branches appear in loops, and they are about 90% taken
- Jumps are about 4–5% of all executed instructions in a program
  - procedure calls are about 1%, and returns another ~1%, of all executed instr.
Branch Taken example

- In modern processors, branch latency is quite long
- In our simple pipeline, branch latency is 2 cycles (read registers; compare)
  (with MIPS–style comparisons (beq/bne only) it could even be 1 cycle)
- Example here with 3–cycle branch latency

need to abort speculative execution before it causes permanent damage: before DM and WB stages

- In this example, each taken branch causes the loss of 3 extra clock cycles
- About 2/3 of all executed branches are taken, so this is a heavy loss
Branch Target Buffer (BTB)

- A small table – a cache, like a hash table – containing pairs of (instruction) addresses for which there is statistical evidence that their next-PC is something other than PC+4

PC of a jump or branch–likely instruction;

Target PC to which this instruction usually went, in the past.

- A ‘best approximation’ – not necessarily correct information
- Branches that are believed not–taken are NOT entered into the BTB
- Like IM –the Instruction Cache– this will oftentimes ‘overflow’: old pairs are removed to make room for more recent ones
- May be complemented with a small hardware stack:
  – on every call (jal ra,...), push the return address;
  – on every return (jr ra), pop an address and predict jumpin to that one

- In parallel with each Fetch, search the fetched instruction’s PC value in the BTB

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>260</td>
<td>200</td>
</tr>
<tr>
<td>40</td>
<td>72</td>
</tr>
<tr>
<td>88</td>
<td>120</td>
</tr>
<tr>
<td>180</td>
<td>160</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
When the BTB prediction is Correct

- When a matching BTB entry is found, use its Prediction; else, fetch from PC+4

- When Prediction is Correct, NO extra clock cycles are lost!
When the BTB prediction is Wrong

- Prediction says: After fetching from 40, fetch from 72
- But this time, the branch ends up going the other way: to 44

- When Mispredicted, branches cost 3 extra clock cycles in this pipeline
Relative Performance

- Define Performance = 1/Execution Time
- “X is \( n \) time faster than Y”

\[
\text{Performance}_X / \text{Performance}_Y = \frac{\text{Execution time}_Y}{\text{Execution time}_X} = n
\]

- Example: time taken to run a program
  - 10s on A, 15s on B
  - \( \text{Execution Time}_B / \text{Execution Time}_A = 15s / 10s = 1.5 \)
  - So A is 1.5 times faster than B
Performance Summary

The BIG Picture

CPU Time = \( \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}} \)

- Performance depends on
  - Algorithm: affects IC, possibly CPI
  - Programming language: affects IC, CPI
  - Compiler: affects IC, CPI
  - Instruction set architecture: affects IC, CPI, \( T_c \)
CPI in More Detail

- If different instruction classes take different numbers of cycles

\[
\text{Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instruction Count}_i)
\]

- Weighted average CPI

\[
\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n} \left( \text{CPI}_i \times \frac{\text{Instruction Count}_i}{\text{Instruction Count}} \right)
\]

Relative frequency
Average Access Time - Caches

- Hit time is also important for performance
- Average memory access time (AMAT)
  - AMAT = Hit time + Miss rate \times Miss penalty
- Example
  - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, L-cache miss rate = 5%
  - AMAT = 1 + 0.05 \times 20 = 2\text{ns}
    - 2 cycles per instruction
Measuring Cache Performance

- Components of CPU time
  - Program execution cycles
    - Includes cache hit time
  - Memory stall cycles
    - Mainly from cache misses
- With simplifying assumptions:

  Memory stall cycles

  \[
  = \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}
  \]

  \[
  = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}
  \]