Reverse engineering micro-architectures

Eric Alata eric.alata@laas.fr

INSA de Toulouse

December 15, 2023

Outline

Outline

Computer Architecture

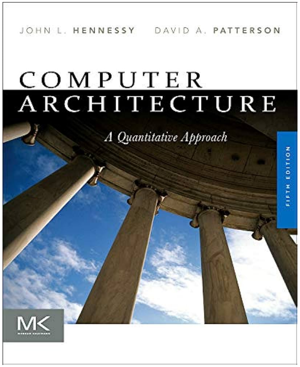
Computer architecture is the science and art of selecting and interconnecting hardware components to create a computer that meets functional, performance and cost goals.

(Mark Hill [Hil])

Textbook

Computer Architecture: *A Quantitative Approach* [HP12]
John Hennessy and David Patterson

- Appendix C
- Chapter 3



ISA

- `sub r0, r1, r2` $\rightarrow$ $r0 \leftarrow r1 - r2$

Simple MIPS architecture [PH13]

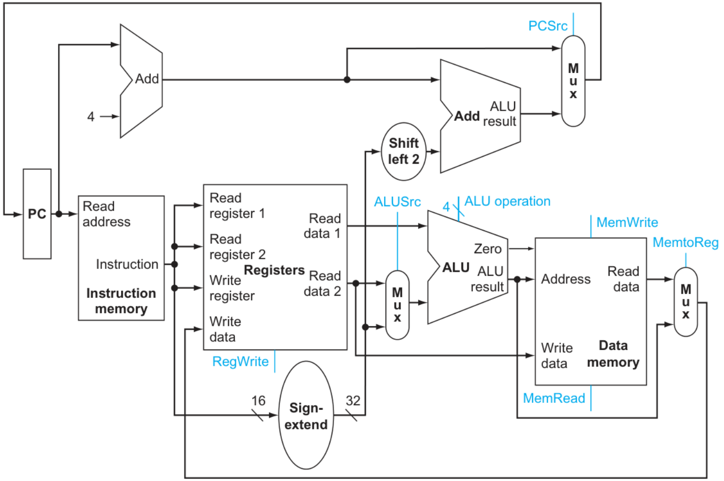


FIGURE 4.11 The simple datapath for the core MIPS architecture combines the elements required by different instruction classes. The components come from Figures 4.6, 4.9, and 4.10. This datapath can execute the basic instructions (load-store word, ALU operations, and branches) in a single clock cycle. Just one additional multiplexor is needed to integrate branches. The support for jumps will be added later.

Pipeline

Classical 5 stages

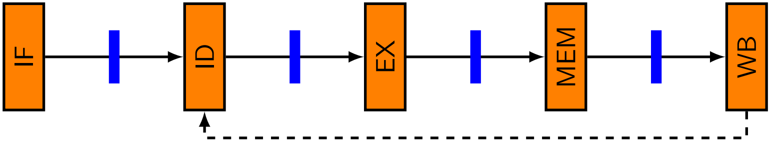
- Instruction fetch – IF
- Instruction decode – ID
- Execute – EX
- Memory access – MEM
- Write back – WB

Pipeline Hazards

- Data hazards
- Structural hazards
- Control hazards

⇒ Forwarding

⇒ Bubble/stall



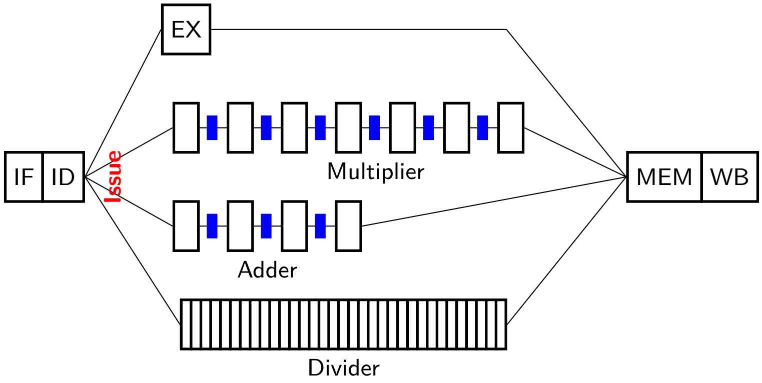
Example 1

- Registers: `ri = i + 10`
- `add r0, r1, r2`
- `add r3, r4, r5`
- `add r6, r7, r8`
- Structural hazard (on registers ID and WB)

Example 2

- Registers: `ri = i + 10`
- `add r0, r1, r2`
- `add r3, r0, r0`

Multicyle pipeline



Latency 3
(w/ forwarding)

`add r0, r1, r2`	IF	ID	A1	A2	A3	A4	MEM	WB			
`sub r3, r0, r5`		IF	ID	ID	ID	ID	A1	A2	A3	A4	MEM	WB

Initiation interval 1

`add r0, r1, r2`	IF	ID	A1	A2	A3	A4	MEM	WB	
`add r3, r4, r5`		IF	ID	A1	A2	A3	A4	MEM	WB
`add r6, r7, r8`			IF	ID	A1	A2	A3	A4	MEM	WB

Example 3

- Registers: `ri = i + 10`
- `add r0, r1, r2`
- `div r3, r4, r5` div by 0?
- `add r6, r7, r8`
- `add r9, r10, r11`
- `add r12, r13, r14`

Multicyle pipeline

- Multiple paths with various latencies
- Multiple instructions may write to the register file at the same time!

$\Rightarrow$ Multiple ports & stall

- Write back stage may proceed instruction out-of-order

$\Rightarrow$ Check registers usage of issued instructions & stall if needed (set in ID, cleared in WB)

- Write-after-write, Read-after-write & write-after-read...

Data Hazards

- Read-after-write (RAW) – True dependence
  i_1: `add r0, r1, r2`
  i_2: `sub r3, r0, r4`
  Swap instructions ⇒ `sub` uses wrong value for `r0`
  Stall pipeline (+ Bypassing)
- Write-after-read (WAR) – Anti dependence
  i_1: `add r0, r1, r2`
  i_2: `sub r1, r3, r4`
  Swap instructions ⇒ `add` uses wrong value for `r1`
  Not in in-order pipeline ; possible in out-of-order execution
- Write-after-write (WAW) – Output dependence
  i_1: `add r0, r1, r2`
  i_2: `sub r0, r3, r4`
  Swap instructions ⇒ wrong value stored in `r0`
- Read-after-read?

Structural Hazards

- Resource used twice in one cycle
- For instance, memory (instruction & data)
- Stall one instruction, usually the younger

Control Hazards

- Must fetch next instruction before the branch outcome is known
- Strategy: always *not-taken* and flush if needed
- Strategy: branch prediction strategy
- Branch Target Buffer

Advanced strategies

- Multiple issue: two or more instructions per stage at once
- Superscalar architectures

In-order pipeline to Out-of-order execution

- Passing functionality to reduce structural hazard
- Use of an instruction buffer
- Dispatch: allocate en entry in the instruction buffer
- Issue: send an instruction from the instruction buffer to the execution unit

Dynamic scheduling

- How many cycles for the execution of the following program?

  i_1: `div r0, r1, r2`

  i_2: `mul r3, r0, r5`

  i_3: `add r6, r7, r8`
- Instruction i_3 does not depend on i_1 and i_2
- It could be executed in parallel...
- Multiple strategies (mainly scoreboard and Tomasulo)
- Focus on Tomasulo algorithm [Tom67]

Tomasulo algorithm

Key elements

- Register renaming
- Instruction queue
- Load/store buffer
- Reservation stations
- Multiple units
- Common data bus

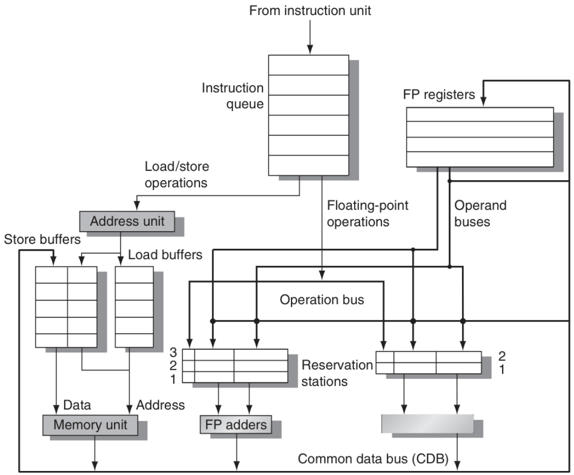


Figure 3.6 The basic structure of a MIPS floating-point unit using Tomasulo's algorithm. Instructions are sent

Speculation [McF93]

- An instruction is (control-)dependant on some set of branches
```
if (a)   ins1; // this instruction depends on a
if (b)   ins2; // this instruction depends on b but not on a
```
- Branch prediction predicts the execution as accurate as possible
- Branch prediction could be wrong (mis-prediction)
- Speculation means to execution instructions according to predictions
- Speculative execution recovery wrong prediction → roll-back execution
- **Remaining traces of misprediction?**
- Useful if the condition is computer after the target address (classical 5 stages?)

Speculation [McF93]

- Types of dynamic branch predictors
  - 1-bit Branch-Prediction Buffer
  - 2-bit Branch-Prediction Buffer
  - Correlating Branch Prediction Buffer
  - Tournament Branch Predictor
  - **Branch Target Buffer**
  - Return Address Predictors
  - Integrated Instruction Fetch Units

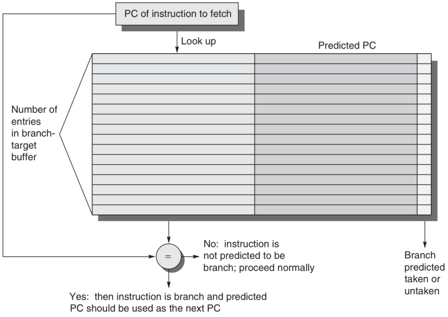
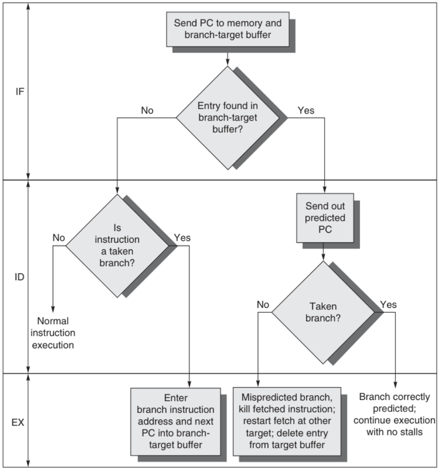

PC of instruction to fetch
Look up
Predicted PC
Number of entries in branch-target buffer
=
No: instruction is not predicted to be branch; proceed normally
Yes: then instruction is branch and predicted PC should be used as the next PC
Branch predicted taken or untaken

Figure 3.21 A branch-target buffer. The PC of the instruction being fetched is matched against a set of instruction


IF
ID
EX
Send PC to memory and branch-target buffer
Entry found in branch-target buffer?
No
Yes
Is instruction a taken branch?
No
Yes
Normal instruction execution
Send out predicted PC
Taken branch?
No
Yes
Enter branch instruction address and next PC into branch-target buffer
Mispredicted branch, kill fetched instruction; restart fetch at other target; delete entry from target buffer
Branch correctly predicted; continue execution with no stalls

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

Tomasulo algorithm

Major issue: precise exception & speculation

- Tomasulo: in-order issue, out-of-order execution and out-of-order completion
- Consider the following example
  i_1: `div r0, r1, r2`
  i_2: `mul r3, r0, r5`
  i_3: `add r6, r7, r8`
- Suppose an Hardware interruption occurs 10 cycles after i_1 is issued
- Instructions i_1 and i_2 have not been committed
- Instruction i_3 has already been committed!
- What about specution? mis-prediction as an exception?

$\Rightarrow$ Use a reorder-buffer out-of-order execution & in-order commit

Outline

Outline

Reverse engineering micro-architecture

- Why *to reverse*?
  - Understand the behavior of the component
  - Produce more efficient software
  - Identify undocumented instructions and behavior
  - Identify weakness (vulnerabilities?)
- What *to reverse*?
  - Almost all elements of the micro-architecture
  - Is the element available in the micro-architecture?
  - Where is this element in the datapath?
  - What are the main dimensions of this element?
- How *to reverse*?
  - Difficulty: we don't have a direct acces to the micro-architecture!
  - We only have acces to ISA
  - Need for a methodology

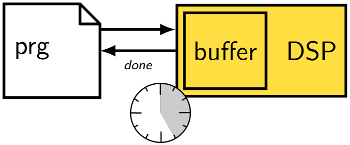
Example

- Target: a DSP used to process data in batch
- The internal memory is finite: batch buffer
- A data can be send to the DSP using the `add <data>` instruction
- Data are internally processed using the `compute` instruction
  And the buffer is flushed
- On `add`, if the buffer is full then `compute` is executed
- Processing time of a data is constant
  $\Delta(\textit{process a data}) = 1 \quad cc$ (cc: clock cycle)
- Processing time of the buffer also depends on an overhead
  $\Delta(\texttt{compute}) = \#\textit{data in buffer} \times \Delta(\textit{process a data}) + 128 \quad cc$
- Processing time of `add`?
- Element under reverse: batch buffer → its size (number of data slots)

How to guess the size of the batch buffer?

Example

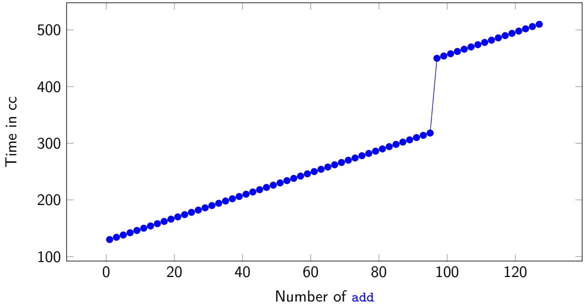
- $\Delta(\textit{process a data}) = 1 \quad \text{cc}$
- $\Delta(\texttt{compute}) = \#\textit{data in buffer} + 128 \quad \text{cc}$
- $\Delta(\texttt{add}) = \begin{cases} \#\textit{buffer size} + 1 + 128 \quad \text{cc}, & \text{if the buffer is full} \\ 1 \quad \text{cc}, & \text{otherwise} \end{cases}$



```python
times = []
for n in range(1, 128, 2):
  execute(["compute"])
  time = execute((["add"] * n) + ["compute"])
  times.append((n, time))
print("n t")
for (n, t) in times: print(n, t)
```

Example

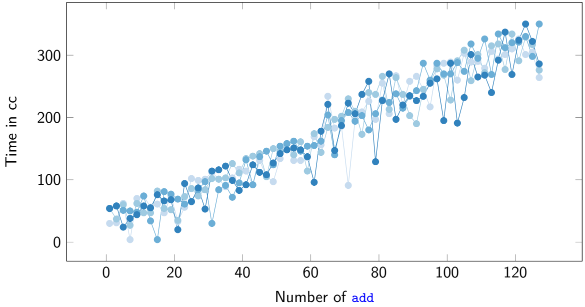
- $\Delta(\textit{process a data}) = 1 \quad \text{cc}$
- $\Delta(\texttt{compute}) = \#\textit{data in buffer} + 128 \quad \text{cc}$
- $\Delta(\texttt{add}) = \begin{cases} \#\textit{buffer size} + 1 + 128 \quad \text{cc}, & \text{if the buffer is full} \\ 1 \quad \text{cc}, & \text{otherwise} \end{cases}$



What is the size of the buffer?

Example

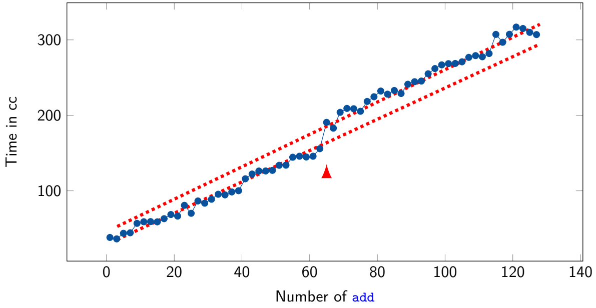
- Real hardware components are more complex
- Branch prediction unit, caches, etc.
- These different units can speed up or slow down treatments
- Makes timing very noisy



What is the size of the buffer?

Example

```
times = [(n, []) for n in range(1, 128, 2)]
for k in range(nb_experiences):
  for i in range(len(times)):
    execute(["compute"])
    time = execute((["add"] * times[i][0]) + ["compute"])
    times[i][1].append(time)
print("n " + " ".join(["t%d" % i for i in range(nb_experiences)]) + " tm")
for (n, ts) in times: print(n, " ".join(map(str, ts)), np.mean(ts))
```



What is the size of the buffer?

Specific topics

- Superscalar architecture
- Scoreboard

References I

- Cs/ece 752: Advanced computer architecture i, `https://pages.cs.wisc.edu/~markhill/cs752/Fall1999`, Accessed: 2023-06-27.
- John L. Hennessy and David A. Patterson, Computer architecture: A quantitative approach, 5 ed., Morgan Kaufmann, Amsterdam, 2012.
- Scott McFarling, Combining branch predictors, Tech. report, 1993.
- David A. Patterson and John L. Hennessy, Computer organization and design, fifth edition: The hardware/software interface, 5th ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2013.
- R. M. Tomasulo, An efficient algorithm for exploiting multiple arithmetic units, IBM Journal of Research and Development **11** (1967), no. 1, 25–33.