Contents

Chapter 1

Micro-Architecture

Micro-Architecture is the way a given ISA is implemented on a processor. Basic blocks of a Micro-Architecture:

Cache A high-speed unit to keep code and data.

- **ICache** holds instructions.
- **DCache** holde data.

IFU A unit to fetch instruction from cache.

IDU A unit to decode instruction after fetch.

EU A unit to execute instruction:

- **ALU** for arithmetic and logic.
- **Branch Unit** for branching instruction.
- **L/S Unit** for loading and saving instruction

Register File A unit to save temporary results.

Program Counter A unit to locate next instruction.

Control Unit A unit to schedule all data movement.

1.1 Standard Benchmarks

1.1.1 Performance Metrics

Latency is the time between start and finish of a single task. The number of taks finished in a give unit of time **Throughput**. **Response time** is the total time to complete a task, also called. Response time consistes of

1. CPU time
   (a) User CPU time
   (b) System CPU time: time spent in OS doing tasks on behalf of a program.
2. I/O time

Then **System Performance** is the inverse time elapsed and **CPU performance** is the inverse user CPU time. From user perespective response time is more important and from system admin perespective throughput is more important.

For CPU time we have:

$$\begin{aligned} \text{CPU time} &= \text{CPU clock cycle} \times \text{Clock cycle time} \quad (1.1) \\ &= \text{CPU clock cycle/Clock rate} \quad (1.2) \end{aligned}$$

To have an estimation for CPU clock cycle we define **CPI**, *cycle per instruction*, to be the average clock cycles per instruction. Thus the Equations (1.1) and (1.2) become

$$\begin{aligned} \text{CPU time} &= \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time} \\ &= (\text{Instruction count} \times \text{CPI})/\text{Clock rate} \end{aligned}$$

Therefore, CPU time is dependent on:

1. Instruction count
   - Program (e.g. Algorithm)
   - ISA
   - Compiler
2. CPI
   - μArch
   - Code CPI also depends on program
3. Clock cycle time
   - μArch and technology

TO evaluate the performance of a computer we use **benchmarks**.

- SPEC Benchmarks
  - CPU/Memory intensive benchmarks
  - some stress CPU
  - some stress memory subsystem
- SPEC Web
  - I/O intensive benchmarks
  - mostly stress I/O subsystem

1.1.2 Speedup

$$\text{Speedup} = \text{Time}_{\text{original}}/\text{Time}_{\text{improved}}$$

Amdahl's law states that speedup is limited to the improved fraction. That is

$$\begin{aligned}\text{Time}_{\text{improved}} &= \text{Time}_{\text{original}} \times \left[(1 - \text{Fraction}_{\text{improved}}) + \text{Fraction}_{\text{improved}}/\text{Speedup}_{\text{improved}}\right] \\ \text{Speedup}_{\text{overall}} &= \text{Time}_{\text{original}}/\text{Time}_{\text{improved}} \\ &= \frac{1}{(1 - \text{Fraction}_{\text{improved}}) + \text{Fraction}_{\text{improved}}/\text{Speedup}_{\text{improved}}}\end{aligned}$$

MIPS, *million instruction per second*, is a performance metric. The drawbacks of this metric are

1. not taking into account capabilites of instructions
2. not realistic metric even on the same CPU

FLOPS, *floating point operations per second*, is another performance metric that measures floating point operations per unit time.

$$\text{FLOPS} = (\text{FP ops/program}) \times (\text{program/time})$$

The drawbacks of this metric are

1. ignores other instruction (e.g. load/store)
2. not all floating point operations have common format
3. depends on how FP-instensive program is

1.2 RTL

Data movement is characterized in terms of

- Registers
- Operations done on registers

RTL, *Register Transfer Language*, describes data movement at register level.

Micro-operations is the functions performed on registers. For example, shift, clear, load, increment, etc. One can describe a μArch in terms of RTL and μOps and control signals that initiate sequence of μOps to perfrom functions.

1.2.1 Register Transfer

Point-to-Point each register has dedicated wires and MUX.

Common input from MUX each register has a load enable and result of the MUX goes to each registe.

Multiple busses something between Common input and point-to-point.

Common bus with output enable similar to common input.

1.2.2 Memomry Transfer

We have **MER**, *Memory Address Register* which enables us to access memory.

Chapter 2

Datapath

Definition (Datapath): A functional unit used to operate on or hold data within a processor. **Contorl unit** is an FSM that schedules data movement in datapath.

Example 2.1. Some datapath elements in MIPS

1. Instruction memory
2. Data memory
3. Register file
4. ALU
5. Adders

PC gets updated automatically which needs another adder so that it always points to the next instruction.

2.1 Datapath elements

2.1.1 Instruction fetch unit

Contains PC which supplies the address for the memory. plus the adder plus the jump adder

2.1.2 Instruction memory

Cause it is slow, we put a cache. takes address from PC and returns an instruction. puts it on IR (note that for a single-cycle it is not a register) decoding happens at the end.

2.1.3 Register file

Also called general purpose registers. Takes 3 indices, two for reading and one for writting, and data to write which is controlled by a signal. returns two data from the reading of the reading indices.

2.1.4 ALU

takes two input from RF (or IR in case of immediate) and returns the result of the operation which is determined be 4-bit select.

2.1.5 Data memory

Also known as D-Cache. two signals for reading and writting. one address input and data writting. returns data read if enabled.

D-cache and I-cache can be :

1. completely separate components
2. seperate components but share bus
3. shared component and bus

the first case it is easy.

2.1.6 Branch unit

for branching which includes sign extend unit, a shifter, and an adder.

Chapter 3

ControlPath

Controller is FSM that given an command outputs the control signals of

1. ALU
2. Multiplixers
3. Registers

Chapter 4

Multi-cycle

The process

Definition:

Cycle 1: IF
- IR <= Mem[PC]
- PC <= PC + 4

Cycle 2:ID & RF
- Calculating control unit
- A <= GPR[IR[25..21]]
- B <= GPR[IR[20..16]]
- ALUout <= PC + (sign-extend(IR[15..0]) « 2)

Cycle 3: Execute
- For BEQ instruction
  - A and B are compared and if zero is set jump is taken by updating PC.
- For R-type instruction
  - ALUout <= A op B
- For Load and store
  - ALUout <= A + sign-extend(IR[15..0])

Cycle 4: Execute
- for R-type instruction
  - GPR[IR[15..10]] <= ALUout
- For store
  - Mem[ALUout] = B
- For load
  - MDR = Mem[ALUout]

Cycle 5: Execute
- For load
  - GPR[IR[20..16]] = MDR

Chapter 5

Pipeline

Principles

1. All instruction must have the same stages in the same order.
2. All intermediate results must be latached at cycle.
3. No functional block reuse for an instruction.

Chapter 6

Memory

We want a fast, large, and a cheap memory. but ofc that is not possible. Principle of locality gets us closer to our goal.

6.1 Principle of Locality

6.1.1 Temporal locality

if you accessed a part of memory, it is very likely you access the same part soon.

6.1.2 Spatial locality

if you accessed a part of memory, it is very likely you access the nearby parts soon. To achieve our goal, we use memory hierarchy, from the big and slow to smaller and faster.

Type	Speed(ns)	Size(byte)
Registers	1s	100s
On-Chip Cache	10s	Ks
Second Level Cache (SRAM)	10s	Ks
Main Memory (DRAM)	100s	Ms
Secondary Storage (Disk)	10s (ms)	Gs
Tertiary Storage (Tape)	10s (sec)	Ts

$$\text{CPU Time} = (\text{CPU execution CC} + \text{Memory stall CC}) \times \text{Clock cycle time}$$

$$\begin{aligned}\text{Memory stall CC} &= \text{reads} \times \text{read miss rate} \times \text{read miss penalty} + \text{writes} \times \text{write miss rate} \times \text{write miss penalty} \\ &= \text{Memory accesses} \times \text{miss rate} \times \text{miss penalty}\end{aligned}$$

$$\text{AMAT} = \text{Hit time} + (\text{Miss Rate} \times \text{Miss penalty})$$

6.2 Cache Structure

6.2.1 Block placement

where a block from a lower level will be placed in the upper level.

Fully associative a block can go anywhere. needs to save the whole address in tag.

Direct mapped a block goes the mod. needs to save $\lg\left(\frac{\text{lower size}}{\text{upper size}}\right)$ bits of the address in tag.

Set associative a block can go anywhere in a set. needs to save $\lg\left(\frac{\text{lower size}}{\text{set size}}\right)$ bits of the address in tag. If each set has 4 block then it is 4-way, if 2, then it is 2-way.

6.2.2 Block identification

how is a block found if it is in upper level. it is dependent on block placement.

$$\text{index} = \lg \#\text{sets}$$

6.2.3 Block replacement

which block should be replaced on a miss.

6.2.4 Write strategy

what happens in a write instruction.

Write-through cache and memory get updated. it is simpler to implement and no need for dirty bit. however it is slower but can be mitigated using write buffers. When allocate-on-write if it is not in the cache gets it and then update both the cache and memory which is faster. No-allocate-on-Write updates the memory if it is not in the cache.

Write-back combines the writes and then write to memory. more complex, needs to implement cache coherency.

6.3 L1 Cache configuration

6.3.1 Split Cache

two indepent caches one for instructions and one for data. easier to read data and instruction at the same time.

6.3.2 Unified

One unified L1 Cache. One can adjust the proportion of instructions and data to fit the program and thus better the hit ratio.