Computer Organization

COMP2120A

Qi Zhao

April 24, 2024

Processor Organization and RISC

Processor Organization

- The processor execute instructions via two operations:
  - moving data from one place to another
  - perform data transformation/processing through ALU.
- The data are stored in registers inside the CPU.
- These operations are controlled by **control signals** generated by the control units,
- Data movement can be described as follows: e.g. Instruction fetch:

```
MAR ⟵ PC
IR ⟵ mem[MAR]
PC ⟵ PC+4
```

Processors

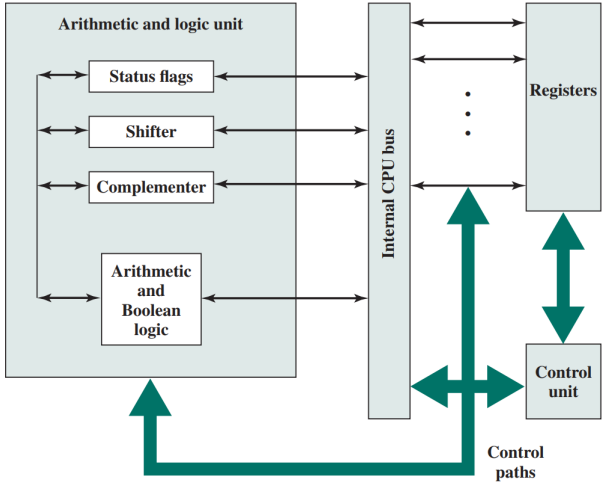


Figure 14.2 Internal Structure of the CPU

Control Signals

- Data movement is controlled by control signals:

  `e.g.` MAR ⟵ PC

  This operation requires 2 control signals:

  1. A control signal to tell the PC to put its content onto the CPU bus
  2. A control signal to tell the MAR to get the value from the CPU bus

- Other operations may require more control signals, e.g. Control signals to tell the ALU what operation to perform, control signals to read/write external memory etc.

Instruction Execution Cycle

- There are 3 stages in an instruction execution cycle:
  1. Fetch Instruction (FI)
  2. Decode Instruction (DI)
  3. Instruction Execution:
     - Calculate Operand address (CO)
     - Fetch Operands from registers/memory (FO)
     - Execute Instruction (EI)
     - Write Operand (WO)
- Note that for modern processors, only `LOAD` and `STORE` instructions are related to memory operands.
- Many instructions don't need CO.
- `LD/ST` instruction do not need to EI

Example

`ADD A, B, C`

where `A, B, C` are direct addressing, i.e. the addresses are in the words following the instruction. This is a 4-word instruction.
Fetch Instruction:

`MAR ⟵ PC`

`PC ⟵ PC+4`

`IR ⟵ mem[MAR]`

Example: ADD A, B, C

Instruction Execute

```
CO1:          MAR  ⟵  PC         # find operand addr
               PC  ⟵  PC+4
              MBR  ⟵  mem[MAR]   # addr of A in MBR
              MAR  ⟵  MBR
FO1:          MBR  ⟵  mem[MAR]   # value of A in MBR
      ALU.input1   ⟵  MBR


CO1:          MAR  ⟵  PC         # find operand addr
               PC  ⟵  PC+4
              MBR  ⟵  mem[MAR]   # addr of B in MBR
              MAR  ⟵  MBR
FO1:          MBR  ⟵  mem[MAR]   # value of B in MBR
      ALU.input2   ⟵  MBR
```

Example: ADD A, B, C

```
EI:    ALU.ouput  ⟵  ALU.input1 +  ALU.input2

CO3:         MAR  ⟵  PC             # write back result
              PC  ⟵  PC+4
             MBR  ⟵  mem[MAR]       # addr of C in MBR
             MAR  ⟵  MBR
WO:          MBR  ⟵  ALU.ouput      # value of C in MBR
       mem[MAR]  ⟵  MBR             # memory write
```

Instruction Pipeline

- How to increase the throughput of instruction execution?
- Divide the execution process into different stages, for example:
  1. Fetch Instruction (FI)
  2. Decode Instruction (DI)
  3. Calculate Addresses (CO)
  4. Fetch Operands (FO)
  5. Execute Instruction (EI)
  6. Write Operand (WO)
- A pipieline

Instruction Pipeline

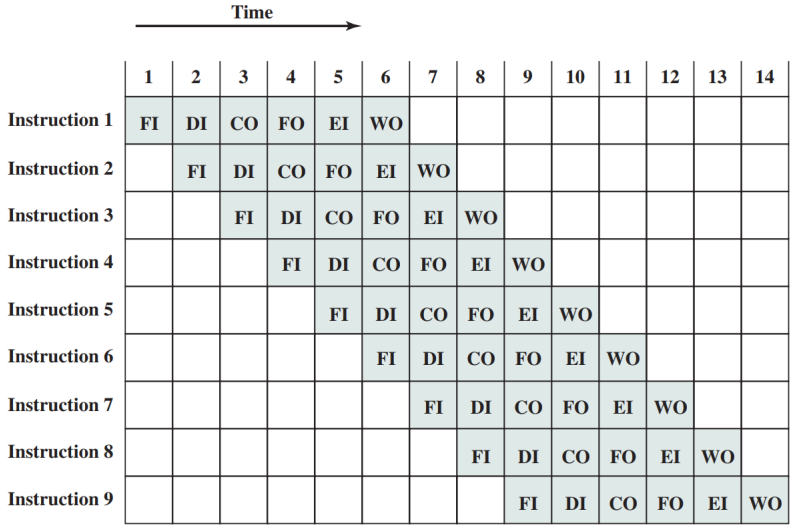
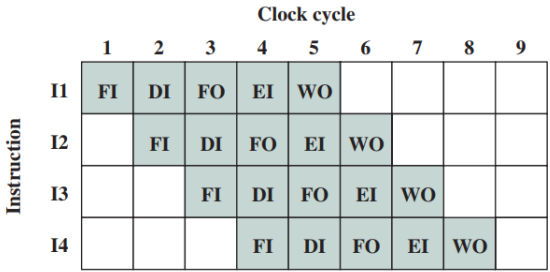


Figure 14.10 Timing Diagram for Instruction Pipeline Operation

Alternative Pipeline

For modern processor, we don't have memory operands. Hence there is no CO stage. We can have a 5-stage pipeline instead.



(a) Five-stage pipeline, ideal case

Pipeline

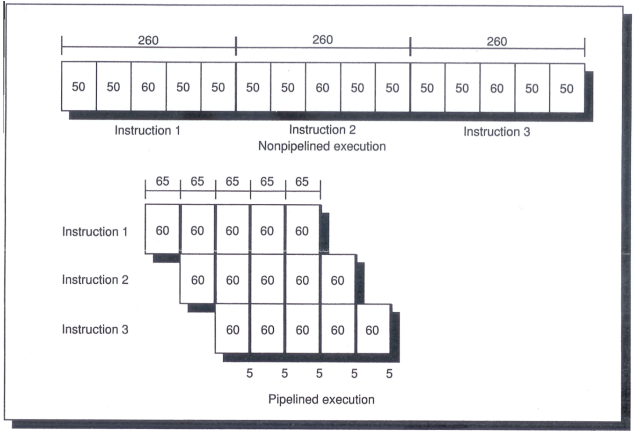


FIGURE 4.6 The execution pattern for three instructions shown for both the unpipelined and pipelined versions. In the unpipelined version, the three instructions are executed sequentially. In the pipelined version, the shaded areas represent the overhead of 5 nS per pipestage. The length of the pipestages must all be the same: 60 nS plus the 5-nS overhead. The latency of an instruction increases from 260 nS in the unpipelined machine to 325 nS in the pipelined machine.

Pipeline

- Note that not all instruction require all stages,
  e.g. `LOAD` and `STORE` only require Memory operation.
- The resources (hardware) required by each stages should not overlap with each other, otherwise duplicate resources are needed:
  - dedicated incrementer for PC instead of using ALU.
  - Separate read port for data and instruction (FI vs FO),
    or may have separate instruction cache and data cache (split cache).
  - multiple internal buses instead of single bus.
- For an ideal pipeline, although each instruction still need to go through all 5 stages (i.e. 5 clock cycles) the throughput is 1 instruction/clock cycle, CPI=1.

Pipeline Hazards

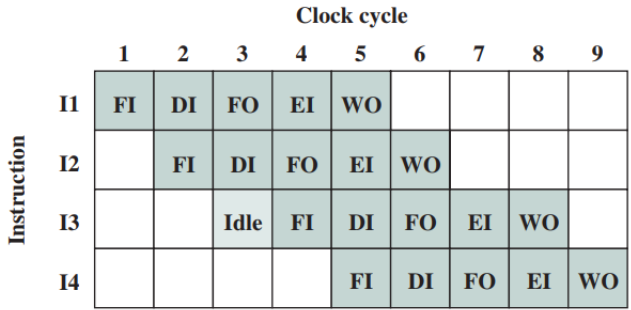
- **Pipeline Hazards** are the situations that prevent the next instruction from entering the pipeline (for execution).
- Three classes of pipeline hazards:
  1. **Resource hazard** — (also called structural hazard) arises from resource conflict, e.g. If only one multiplier unit available, and current and next instruction both require multiply, the next instruction must wait for previous multiply to finish, e.g. when PC increment and ALU operation both requires the ALU, there will be resource conflict.
  2. **Data hazard** — instruction waiting for the result of previous instructions. (data dependency)
  3. **Control hazard** — arises from branches or other instructions that changes the PC (e.g. CALL/JSR, jump subroutine). We don't know where to continue until the branch instruction finishes.

Resource Hazard

- For example, if only one memory read port is available, then instruction fetch and memory read/write cannot be overlapped. (Note that not all instruction require memory read/write).
- For example, the increment of `PC` cannot be overlapped with ALU operations, if both requires ALU.
- Solution: Add more resources, e.g. Dedicated incrementer for `PC`,
  separate data/instruction cache (split cache), so that `FI` and `FO` stage will use different hardware.

Resource Hazard

Here we assume that the source operand and instruction are in memory, while the destination operand is in a register.



(b) I1 source operand in memory

Data Hazard

- Example:
  ```
  ADD R2, R3, R1
  SUB R1, R5, R5
  ```
- source operand `R1` of the `SUB` instructions depends on the result of previous instruction.
- To solve this problem, you can re-arrange the instruction (if possible) so that `R1` has already been written back to the register file, before instruction `SUB` start fetching its operand.
- You may need to insert two or more instruction in between, depending on how the pipeline is arranged and optimized.

Data Hazard

```
ADD EAX, EBX # EAX = EAX + EBX
SUB ECX, EAX # ECX = ECX - EAX
```

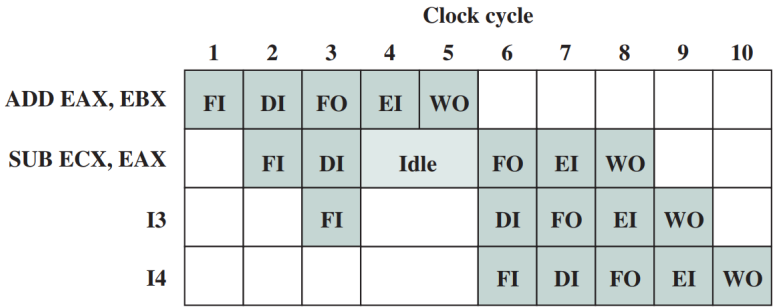


Figure 14.16 Example of Data Hazard

Data Hazard

- For example:

```
ADD R11, R12, R13          ADD R2, R3, R1
ADD R14, R15, R16    ->    ADD R11, R12, R13
ADD R2, R3, R1             ADD R14, R15, R16
SUB R1, R5, R5             SUB R1, R5, R5
```

- There is also hardware solution, by forwarding the result of `ADD` from the output of the ALU to the input of ALU when `SUB` is executed. (This is called data forwarding)

Types of Data Hazard

Consider two instructions I_1 and I_2 in sequence.

- Read after Write (RAW) Hazard — I_1 writes to a register and I_2 reads the same register. Hazard occurs if the read is before the write.
- Write after Read (WAR) Hazard — I_1 reads a register and I_2 writes the same register. Hazard occurs if the write is before the read.
- Write after Write (WAW) Hazard — I_1 and I_2 both writes to the same register, and I_2 writes before I_1.
- Only RAW hazard occurs in a pipeline. (Each stage of the later instruction will only be dispatched after the same stage of the previous instruction finishes.) The others (WAR and WAW) occurs in parallel systems.

Control Hazard

- The `FI` stage of the next instruction cannot start until the branch is resolved.
- the CPU must wait, and there is no other way to resolve it.
- One can use Branch Prediction.
  - On a branch instruction, we continue execution along one of the two paths.
  - 50% of the time we will choose the correct path.
  - In some situation, e.g. branches corresponding to for-loop, we will have even higher correct prediction (because in a for-loop, most of the time, we will branch back).

Control Hazard

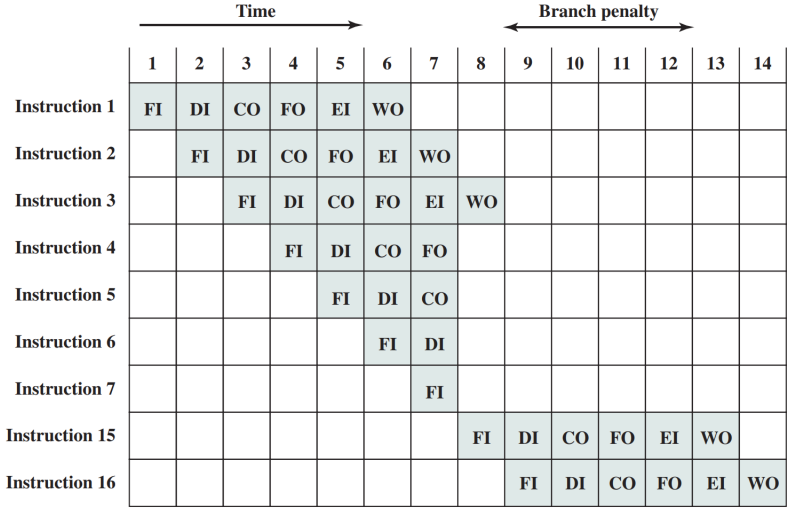


Figure 14.11 The Effect of a Conditional Branch on Instruction Pipeline Operation

Dynamic Branch Prediction

- For simple branch prediction, we will change our next prediction when there is a wrong prediction.
- However, this may not be desirable, and we may want to change only when there is consecutive two wrong prediction.
- Rationale: For example, in a for-loop, usually, we will predict branch taken. However, for the last iteration, the branch is not taken. The next time, branch is taken again. We have a wrong predict.
- Dynamic Branch Prediction: require two consecutive wrong prediction to change decision.

Dynamic Branch Prediction

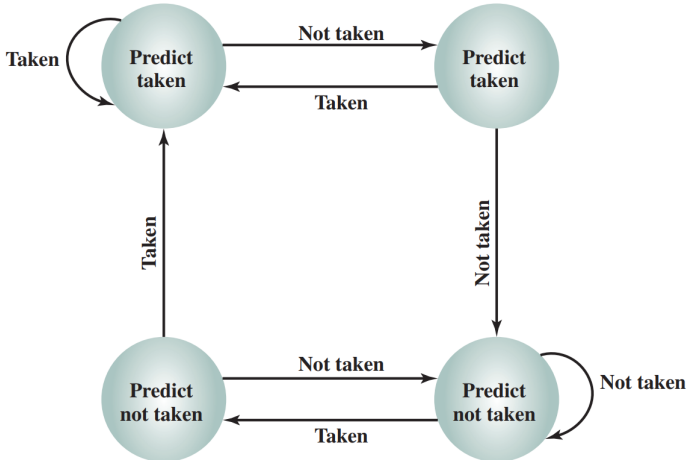


Figure 14.19 Branch Prediction State Diagram

Processor Performance

$$\text{Execution Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

where $\text{CPI} = \text{Clock per instruction}$.

- Clock rate =1/clock cycle time
- Is a 4GHz processor better than a 3GHz processor ?
- Clock rate not the only factor, when comparing two processors.
- You have to compare how many instruction is needed to achieve a function, and how many clock cycle an instruction requires.

Processor Performance

- Exploit more effective instruction pipelining. This will lead to a smaller value of CPI. (The throughput of instruction execution is higher with pipelining).
- use multiple instruction execution unit.
- large register file — reduce memory access, hence increasing speed
- simplified instruction set — reduces or eliminates the need for microprograms, which is slower than hardwired implementation

Characteristics of Modern Processors

- All instructions are register-register type except `LOAD` and `STORE` (which access memory)
- Fixed length and simple, fixed format instructions — for example, we can start fetching register operands if they are always at the same place within the instruction and discard the read if there are no operands.
- Relatively few operations and addressing modes — Simple CPU, simpler CPU implementation, faster clock rate. Also, the pipeline can be designed more efficiently, if the are less cases to consider.
- Use of hardwired rather than microprogrammed control.
- Use of instruction pipelining and extensive software and hardware techniques to eliminate the effect of pipeline hazards.
- These are concepts of Reduced Instruction Set Computers (**RISC**).

Performance Improvement by using Registers

```
ADD A, B, C
```

```
LD A, R1
LD B, R2
ADD R1, R2, R3
ST R3, C
```

Number of Memory Access = 3 (in both cases)

Performance Improvement by using Registers

```
ADD A, B, C
ADD D, C, F
ADD B, F, C
```

```
LD A, R1
LD B, R2
ADD R1, R2, R3
LD D, R4
ADD R4, R3, R5
ADD R2, R5, R3
ST R3, C
ST R5, F
```

Number of Memory Access = 9 vs 5

Performance Issue

- Simple Instruction Set, and hardwired logic
  $\longrightarrow$ high clock rate (low clock cycle time)
- Extensive Pipelining
  $\longrightarrow$ low CPI
- Simple instruction
  $\longrightarrow$ high instruction count
- However, empirical studies shows that the increase in instruction count is usually very small, e.g. 20% even using simple instruction.
- Hence there is an overall improvement in performance

Relative Code Size

Three studies were performed to compare the number of instructions executed in different sets of C programs for different processors, including RISC I processor (reduced instruction set computer), against complex instruction set processors (VAX-11, Motorala M68000, and Zilog Z8002).

Table 15.6 Code Size Relative to RISC I

	[PATT82a] 11 C Programs	[KATE83] 12 C Programs	[HEAT84] 5 C Programs
RISC I	1.0	1.0	1.0
VAX-11/780	0.8	0.67	
M68000	0.9		0.9
Z8002	1.2		1.12
PDP-11/70	0.9	0.71	

- The instruction count only increases by 20% – 30%.
- CPI reduced by 2-4 times. (e.g. VAX-11/780 CPI=8)
- Clock cycle time is also reduced.

Characteristics of RISC

- All instructions are register-register type except LOAD and STORE (which access memory)
- Fixed length and simple, fixed format instructions.
- Relatively few operations and addressing modes.
- Use of instruction pipelining, instruction level parallelism and extensive software and hardware techniques to eliminate pipeline disruption.
- Rely on optimizing compiler to enhance system performance.

Use of Large Register File

- Because of the simplicity of the CPU, can have chip space for a large register file.
- Register can be used as a cache for variables. Instructions using registers are much faster than instruction with memory operands.
- Compiler will do the this mapping between register and variables — **Register Allocation**.
- Register Allocation is one of the most important optimization techniques in modern compiler.

RISC

- Note that these measures mentioned above can be applied to any processor design, and not limited to RISCs.
- Actually, RISC is a design philosophy where performance of CPU is enhanced.
- Nothing prevents a CPU designer from incorporating these measures into their CISC (Complex Instruction Set Computer).
- Nowadays, usually we will see some kind of hybrid design.