Computer Architecture
(计算机体系结构)

Lecturer: Jingwen Leng

Note Taker: Yihang Qiu

Introduction

数字电路—微体系结构（micro architecture）

—汇编语言（Assemlbly）—低层高级语言—编程框架—高层高级语言—AI 模型

| ———————— Architecture （ISA） ———————— |

【计算机的系统栈】

Bits → Gates → Processor → Instructions → C Programming

#1 Universal Machine（通用）

Turing Machine

Mathematical device of a device capable of performing all kinds of computations proposed by Alan Turing.

1) read and write symbols on an infinite "tape".
2) state transitions based on current states and symbols

Universal Turing Machine

A machine implementing Turing Machines. (Also a Turing Machine)

Programmable

In Practice: Time Limit, Memory Limit

#2 Transformations between layers

Devices→Circuits→Microarchitecture→ISA→Program→Algorithm→Problem

(Instruction Set Architecture)

Data Representation and Operation

1） "1" – presence of voltage; "0" – absence of voltage [Digital] ←→ [Analog] Complex to recover

2） Binary System *omitted*

3） Unsigned Numbers – weighted positional number (in binary system) [MSB, … , LSB]

Addition, Subtraction, Multiplication, Division, carry, *etc.* *omitted*

4） Signed Numbers – add a "sign bit".

Sign-magnitude; 1's complement; 2's complement （原码，反码，补码）*omitted*

(complex in computations) (two forms of "0")

n bit w. a signed bit: $-2^{n-1} \sim 2^{n-1}-1$

5) Sign Extension（符号位扩展）: **pad the sign bit**

e.g. 0100(+4) → 00000100(+4)

1100(-4) → 11111100 (-4)

6) Overflow Test

举个例子就好了。

· Sign of both operands are the same but different from the sign of the sum.

· Carry into MSB does not equal carry-out.

e.g. 01000+01001 == 10001 IN: 0 OUT: 1(sign bit)

10001+11111 == 00000 IN: 1 OUT: 0(sign bit)

7) Logical Operations: *omitted*

NOT, AND, OR

8) **Hexadecimal Notation:** not a new representation, but a simple way to note it.

9) **Fractions:** Precision and Range 精度和范围

Fixed-point（定点数）: *omitted* (same as integer, 2's comp addition still work) 精度相同

Floating-point: $F\times2^e$ (fraction, exponent)

IEEE 754 std. [S]+[exp, 8b]+[frac, 23b] = 32 bits <single float>

$$N = \begin{cases} (-1)^S \times 1.\text{frac} \times 2^{\text{exp}-127}, & 1 \le \text{exp} \le 254 \\ (-1)^S \times 0.\text{frac} \times 2^{-126}, & \text{exp} = 0 \\ used\ to\ represent\ \text{NaN, INF, etc.}, & \text{exp} = 255 \end{cases}$$

When exp = 0, 1.xxx×2^(-127) → 0.1xx×2^(-126)

('cause we want 0000000000 to be 0)

e.g. [1 011111101 00000000000000000000000] = $-1.0\times2^{126-127}$ = -0.25

exponent = 127 → [1,2) res. ratio = 1/(2^23)

exponent = 128 → [2,4) res. ratio = 2/(2^23)

The precision ratio is "floating" as the point floats. 精度变化

Nearer to 0, more precise.

Floating-point Operations: normalize → calculate → represent.

10) ASCII

Capitalized: ascii-$(20)_H$ 全转小写/大写：OR/AND $(010\ 0000)_2 = (20)_H$

※ Unicode

11) LC-3

16-bit Data Types: 2's comp. signed integer, ADD, OR, NOT

Digital Logic

1) Transistors

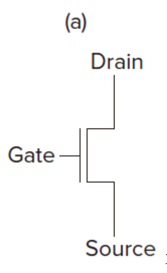
Logically, each transistor acts as a switch.

$\rightarrow$ combined to implement logic functions. AND/OR/NOT

MOS: Metal-Oxide-Semiconductor

N-type MOS: Gate==1(positive voltage) open / Gate==0(zero voltage) closed

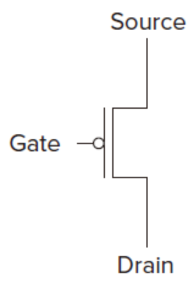
Pull out DOWN when input=1 (reverse property)



must be connected to GND

P-type MOS (complementary to N-MOS): Gate==0 open / Gate==1 closed

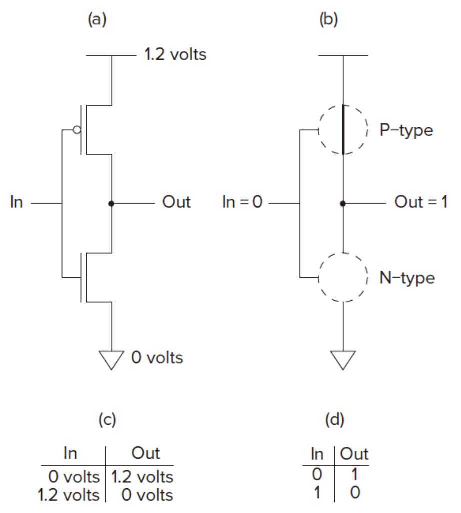
Pull out UP when input=0 (reverse property)



must be connected to high voltage

0:[low voltage range] illegal [high voltage range]:1

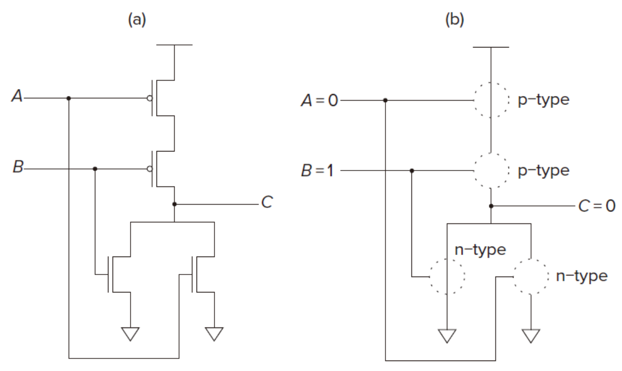
CMOS: Use both N-MOS and P-MOS



In	Out
0 volts	1.2 volts
1.2 volts	0 volts

In	Out
0	1
1	0

(Inverter, NOT Gate)



(c)

A	B	C
0 volts	0 volts	1.2 volts
0 volts	1.2 volts	0 volts
1.2 volts	0 volts	0 volts
1.2 volts	1.2 volts	0 volts

(d)

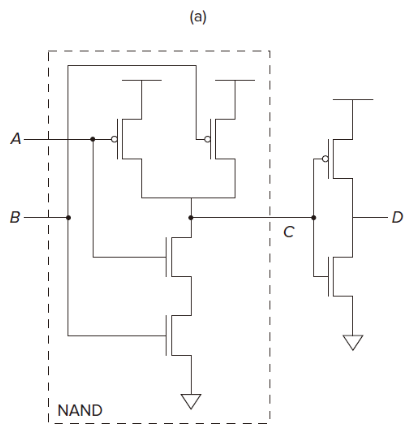
A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

Figure 3.5 The NOR gate.

OR = NOT NOR

Serial P-MOS on top, parallel N-MOS on bottom | | NOR cascade a NOT

** See 3) De Morgan's Law*



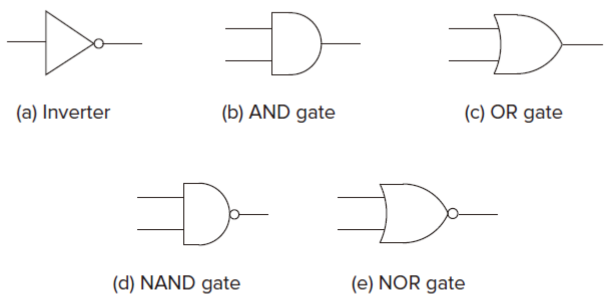
(b)

A	B	C	D
0	0	1	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 3.8 The AND gate.

AND = NOT NAND

2) Basic Logic Gates: *omitted*



3) De Morgan's Law *omitted* $\overline{A+B} = \bar{A}\bar{B}$

The structure of NOR Gate: serial P-MOS ($\bar{A}\bar{B}$), parallel N-MOS ($A+B$)

The structure of NAND Gate: parallel P-MOS ($\bar{A}+\bar{B}$), serial N-MOS (AB)

4) History of CMOS

TTL Circuits (partial) (P-MOS only)

Problem in TTL: <u>Low working efficiency</u> constrained by the Resistance.

The charging time will be effected by Resistance ($\tau = RC$), we want smaller Resistance without the risk of burning the wire (since $I = U/R$).

In CMOS, no chance of short connect and no Resistance → high frequency, GHz level

5) customized Gates

2 std. AND Gates: 16 transistors (for 3 inputs)

1 cus. AND Gate: 8 transistors (for 3 inputs)

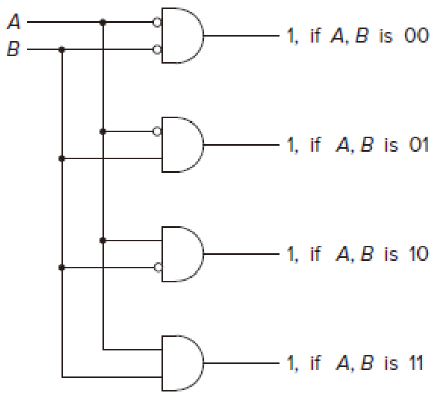
6) Building Functions: *<u>omitted</u>*

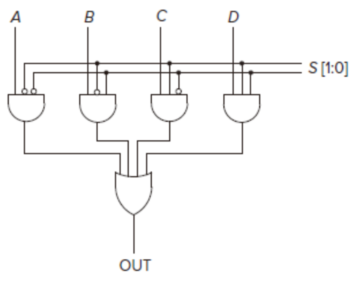
Combinational（组合） / Sequential（时序）

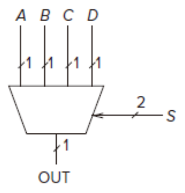
w/o state, Outputs = f(Inputs) w. states Outputs = f(Inputs, States)

7) Decoder, Multiplexer (MUX), : *<u>omitted</u>*

(a)







←/—: n bit inputs

n

8) Sum of Product / Product of Sum

9) Full adder: *<u>omitted</u>*

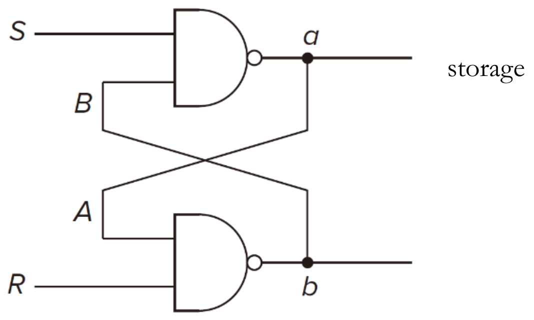
Ripple-carry Adder (the propagation time is accumulated)

Lookahead Adder

10) Logical Completeness

Sequential Circuit

11) Latch: R-S Latch -- can use truth table to prove its “latching” property.



R==0: stg=0 [RESET]

S==0: stg=1 [SET]

R==1 && S==1: stg holds original value [LATCH]

R==0 && S==1: stg and ~stg both are "1". Next state unknown. （次态不定） [ILLEGAL]

12) D-Latch:

Inputs: D, WRITE ENABLE (WE) (S == ~(D ∧ WE), R == ~(~D ∧ WE))

WE = 1, out = D； WE = 0, out = LATCH

13) [Adder] → [Register（寄存器）] (multiple D-Latch)

14) A[14:9]: represent the 14th-9th bit of A (index start from 0)

15) Memory（内存）: logically viewed as a table divided into multiple lines (a line – a datum stored)

Address. → a particular "line" (location)

Addressability: # (bits per location)

Usually, =1 Byte = 8 bits

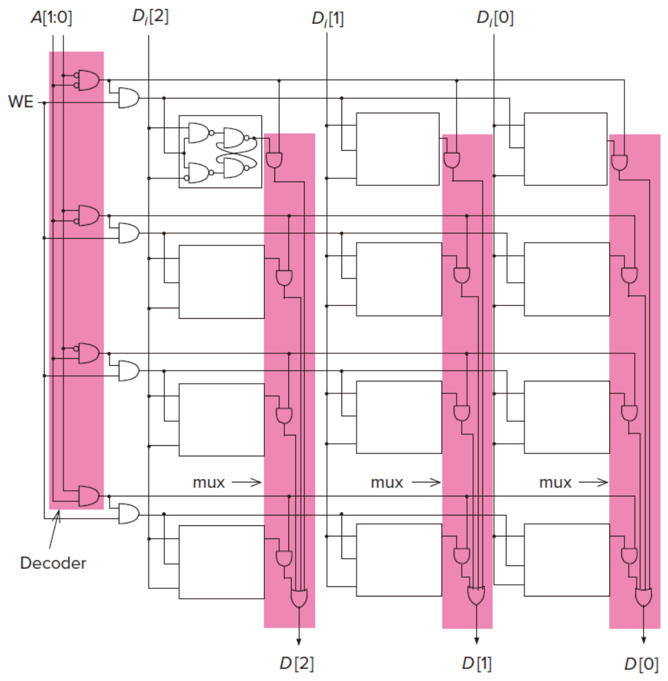


Figure 3.20 A 2^2-by-3-bit memory.

Moreover, we can add “READ ENABLE”

→ RAM（随机存储器） and ROM（Read-Only Memory 只读存储器）

RAM: the time of the visit to a certain address is unrelated to the address.

Data Will Eliminate without Power Maintain — Volatile storage (易失存储)

SRAM (Static Random Access Memory) and DRAM (Dynamic RAM)

e.g. tape（磁带） is not a RAM.

输入：地址(n-bit address)+数据(m-bit data)+1 位读写控制

输出：m-bit data

SRAM < Latch（晶体管数量更少）→ Faster, maintain data as long as power applied

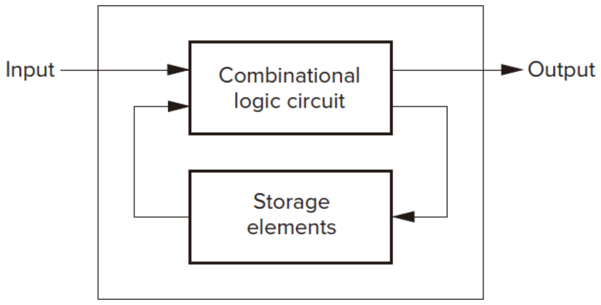
DRAM：一个晶体管 Capacitance holds the data voltage. → data decays as time

Slower, denser, need to be refreshed (capacitance!!!)

Other Storages: disk, flash, etc.

External Storage：Non-volatile data

16) State Machine（状态机）



17) State Diagram: *omitted*

18) Finite State Machine:

Finite states, finite external inputs and outputs

Explicit specification of all state transitions and what determines each external output value

19) The Clock

A clock circuit triggers the transition. Rising-edge, falling-edge

20) Level-triggered（水平触发）: e.g. Latch (ENABLE=0/1 → different states)

Edge-triggered: e.g. Master-slave Flip-flop（主从触发器）

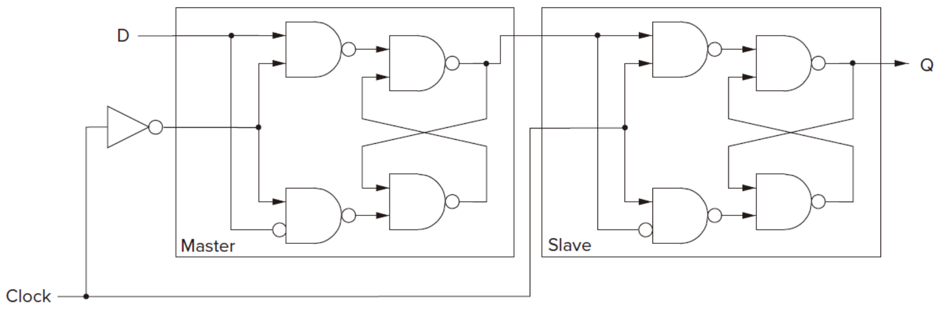


Figure 3.33 A master/slave flip-flop.

Computer Organization and ISA

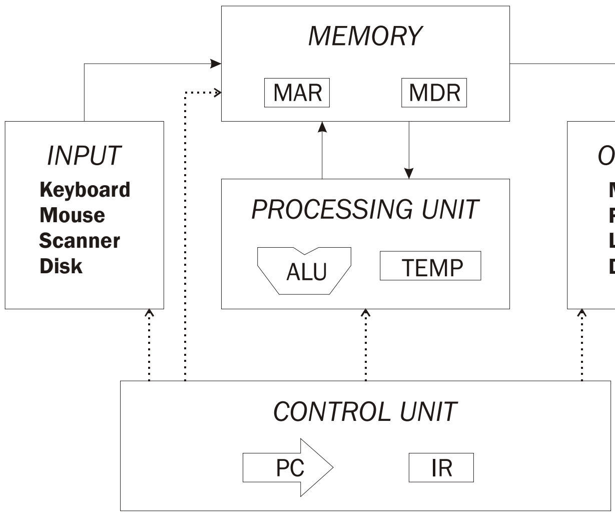
ENIAC (hard-wired program) → EDVAC (program stored in memory) (binary system applied)

→ von Neumann Model（冯·诺依曼结构）

Von Neumann Model

memory + processing unit + control unit + input/output device

(Programs stored in Memory ← Need a Control Unit (No direct data interaction, assistance only))



(Each component can be viewed as an FSM)

Solid Lines: Data Exchange (or single-way feed-in)

Dotted Lines: Assistance, no actual data exchange

Memory: Basic Operations (LOAD, STORE)

Interface（接口） to get data to/from the memory → <u>Register（寄存器）</u>

- MAR: Memory Address Register
- MDR: Memory Data Register

Need a datum from the memory? – LOAD

1) Write the address into MAR; 2) C.U. send a "read" signal to Mem.

3) Read the data from MDR.

Need to store a datum into the mem.? – STORE

1) Write the address into MAR; 2) Write the data into MDR;

3) C.U. send a "write" signal to Mem.

Processing Unit:

ALU = Arithmetic and Logic Unit

TEMP = Registers（通用寄存器）, as temporary storage

-- MEM's read-and-write speed is much lower

TEMP is faster to visit (compared with MEM). (can complete in a single machine cycle)

Can store: data/results from ALU/address/instructions/ etc.

Word Size（字长）: #(bits normally processed by ALU in one instructions),

一次指令能处理的数据的位宽, **e.g.** addition
also width of registers in TEMP

Inputs and Outputs

Devices which are both input and output: Disk, Internet, etc.
Program controlling access to a device is a ***driver***.

Each device has its own interface, usually a set of registers like MAR and MDR.
e.g. Keyboard – KBDR (data register), KBSR (status register)
Monitor（显示器）-- DDR (data register), DSR (status register)
*Data register is not enough, because we don't know whether there is data in KBDR/DDR or not.
(KBSR: to tell the computer whether all-0-coded inputs **vs.** no data inputted)
(DDR: monitor busy **vs.** can be fed in data)

Control Unit

IR: Instruction Register – contains the current instructions [VALUES]
PC: Program Counter – contains the address of the next instruction to be executed. [PTR][1]

Read an instruction from the MEM.
Interpret the instructions, generate signals to tell other components what to do
(an interpretation might take several rounds to complete)

Instruction Coding

Instruction

· Opcode（操作码）: operation to be performed. E.g. addition
· Operands（操作数）data/**locations** to be used in the op. numbers to be added

Instructions: Encoded as bits. (Often of fixed lengths) (not all, e.g. x86 and x64)
An instruction: completed or not executed at all.

ISA: Instruction Set Architecture, **i.e.** instructions and their formats
Software → ISA → Hardware
后向兼容（backward compatibility）: after update, the old instructions still can work
Software on a different ISA need to be redesigned. (Cater to the ISA)
LC-3: [15:12] – opcode (16 opcodes in total)
[11:9] – destination register storing the final result
If some part won't be used in an instruction, there must be a formal way to code it (all "0"s or all "1"s, for example).

CALCULATIONS
[8:0] – source registers providing operands (3 operands at most)
LOAD
[8:6] – base (which register) ["PTR"]
[5:0] – offset ["VALUE"] (immediate number，立即数)

[1] **PTR:** pointer

Add the offset to the contents of base $\rightarrow$ form an address

Instruction Process

A larger FSM.

1) fetch an instruction from the MEM.	[PC in U.C. tells]
2) decode the instruction	[store to IR]
3) evaluate the address (if need data from MEM)	[store to MAR]
4) fetch operands from MEM	[from MDR]
5) execute operation	[P.U.]
6) store result	[store into MEM or registers]
7) go to 1).	

Use [·] to denote the content of ·.

Fetch

1) Load the next instruction (at the address stored in PC) from MEM into IR
   - Copy [PC] to MAR
   - C.U. send "read" to MEM.
   - Copy [MDR] to IR
2) PC ++

Decode

1) A decoder asserts the control line corresponding to the desired opcode.
2) Identify other operands.

Evaluate Address (Optional)

Calculate the address used for access. (e.g. offset to MDR/PC/etc.)

Fetch Operands (Optional)

Execute (Optional)

Send operands to ALU to complete the calculation or do nothing

Store Result

Destination could be a register or some part in the MEM.

Changing the Sequential Instructions

e.g. Loop, if-then (branches), function call

Control Instructions

- **Jump:** directly change the [PC] (the operand of "Jump" is PC) -- unconditional
  e.g. in LC-3, [JMP] [000] [BASE] [000000], default DST is PC.
- **Branches:** change the [PC] only if some conditions are true -- conditional

Category of Basic Instructions:

1) computational instructions; (ADD/MUL/NOT/AND/OR, etc.)
2) data movement instructions; (LOAD/SOTRE, etc.)
3) control instructions. (JMP/BRnz, etc.)

The state diagram of LC-3: *Introduction to Computing System: From Bits and Bytes* Page.702

* Stop the CLK – operations like fetch operands from MEM might take longer time than other steps.

ISA

All the programmer-visible components and operations

- Memory organization -- address and addressability
- Register set:
  - IR, MAR, MDR is not a part of ISA: can't be visited and updated directly
    (Their content only change iff. Instruction requires)
  - On the other hand, PC is a part of ISA. (JMP can change its content directly.)
- Instruction Set (opcodes, data types, instruction addresses)

~ Machine Language

LC-3

MEM:

In LC-3, addressability（寻址颗粒度） is 16 bits.

** In reality, addressability is often 8 bits.

Register in Processing Unit: R0~R7

15 opcodes. 16-bit 2's complement integer.

Condition codes ← used in Control Instruction BRANCH. e.g. Negative, Zero, Positive

Addressing Modes（寻址模式）：

1) Non-memory address: register's index
2) Memory address: PC-relative (base + offset), PTR, etc.

Operation Instructions

3 ops only: NOT, AND, ADD.

Operands: Registers

e.g. **NOT** [1 0 0 1][DST][SRC][1 1 1 1 1 1] ([5:0] is not used)

[[Register Files]] SRC → [[ALU]] → DST [[Register Files]]

e.g. **AND** [0 1 0 1][DST][SRC1][1][Imm (5 bits)]

↑ _immediate number

[[Register Files]] SRC1 / SRC2 → [[ALU]] → DST [[Register Files]]

AND [0 1 0 1][DST][SRC1][0][0 0][SRC2] ([5:4] is not used)

↑ _Register

[[Register Files]] SRC1 ——————————————————→ [[ALU]] → DST [[Register Files]]

[[SExt]] (Sign bit Extension) → 16-bit 2's complement →↑

↑

[[IR]] IMM == IR[4:0]

ADD is similar to AND, has two different modes (IMM/Register)

How to get OR? De Morgan's Law, with **NOT** and **AND**

How to subtract? **(1) NOT** R2→R2, **(2)** R2+1→R2, **(3)** R1+R2→DST

How to copy? R1+0→R2

How to clear? R1 **AND** 0→R1

Movement Instructions

Load -- read data from memory to register

- LD: PC-relative mode
- LDR: base+offset mode
- LDI: indirect mode

Store -- write data from register to memory

- ST: PC-relative mode
- STR: base+offset mode
- STI: indirect mode

Load effective address -- compute address, save in register

- LEA: immediate mode
- *does not access memory*

1) PC-related Addressing Mode (LD/ST)

Base = &PC

Offset: 9-bits, -256<=Offset<=255

LD: [0 0 1 0][DST][PC-OFFSET (9 bits)]

[[PC]] → **[[ADDRESS ADD]]** → [[MAR]] MEM→ [[MDR]]→DST [[Register Files]]

SExt →↗**(aside from ALU ADD)**

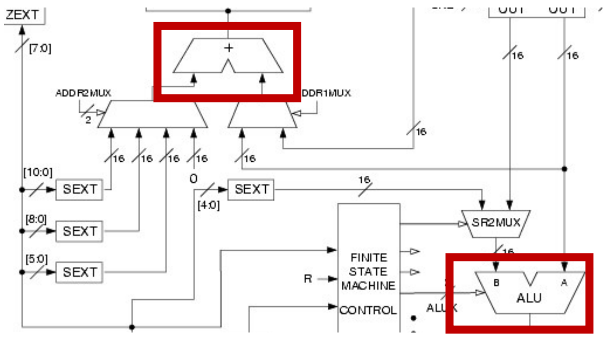
(IR [8:0] converted into 16 bits 2's-complement)

↑

[[IR]]

Expression: (only in this note)

&[PC+IR[8:0]]→DST[2]



2) Indirect Addressing Mode (LDI/STI)

Base = @PC.

The address of PC-related address stores the real address we need to visit. (firstly visit a PTR, then visit the location using the PTR)

LDI [1 0 1 0][DST][PC-OFFSET(9)]

Similar to PC-related address, after get the first address, [MDR]→MAR MEM→ MDR→DST

Expression: (only in this note)

2 Use @· to denote the address of · .
Use · to denote the content of · .
Also, ·[x:y] denotes the content from x-th-bit to y-th-bit of · .
M[·] denotes the content of address · in the MEM, also denoted by &[·] in this note .

&[&[PC+IR[8:0]]]→DST

3) Base + Offset Addressing Mode (LDR/STR)

Base = @Registers (in P.U.), similar to PC-related.

LDR [0 1 1 0][DST][BASE][OFFSET(6)] &[BASE+IR[5:0]]→DST

4) LEA（立即数寻址）

LEA [1 1 1 0][DST] [OFFSET(9)] PC+IR[8:0]→DST

e.g. &PC = [x30F6] decoded: LEA, OFFSET=-3, DST=1

at this time, &PC has already move to the next space x30F7, x30F7-3 = x30F4 → R1

std. denotation: R1←PC-3=x30F4

Note that PC has already be incremented(++) by FETCH stage!

Control Instructions (JUMP, BRANCH, TRAP)

LC-3 Condition Codes: N—negative, Z—zero, P—positive (exactly one would appear)

(set by any instruction writing a value to a register)

BRANCH → jump to a instruction within 256 of the BR-instruction.

BR [0 0 0 0][n][z][p][PC-OFFSET(9)]

e.g. ⩾0 n = 0, z = 1, p = 1

IR[8:0]→Sext→AddressADD ——→ ↓

PC↗_---------——→ PCMUX → PC

IR[11:9] → LOGIC（ALU）control ↑

Condition codes: Load instructions (LD, LDI, and LDR) and operate instructions (ADD, AND, and NOT) each load a result into one of the eight general purpose registers. The condition codes are set, based on whether that result, taken as a 16-bit 2's complement integer, is negative (N = 1; Z, P = 0), zero (Z = 1;N, P = 0), or positive (P = 1;N, Z = 0).

Condition codes Three 1-bit registers: N (negative), Z (zero), and P (positive). Load instructions (LD, LDI, and LDR) and operate instructions (ADD, AND, and NOT) each load a result into one of the eight general purpose registers. The condition codes are set, based on whether that result, taken as a 16-bit 2's complement integer, is negative (N = 1; Z, P = 0), zero (Z = 1; N, P = 0), or positive (P = 1; N, Z = 0). All other LC-3 instructions leave the condition codes unchanged.

For-loop: iteration index→Register→[n/z/p] 12 iteration, then i←12, i-- til 0.

JUMP → unconditional branch

JMP [1 1 0 0][0 0 0][BASE][0 0 0 0 0 0] JMP to the content of BASE Register

TRAP: calls a service routine. 类似函数调用

Jump to the address storing some operations given by the system.

(similar to "jump to the codes of a function")

Sentinel: special character used to indicate the end of a sequence (a file, a string, etc.)

Data Path

Introduction to Computing System: From Bits and Bytes *Page.704*

Tri-state Gate: 0, 1(通路，传输数据), 断路 → used when components need to "write" into the bus（总线）.

ALU: inputs from REG FILE and IR
output – into the bus
REG FILE: inputs: DR, SR1, SR2, from bus
outputs: 2 outputs (into ALU, in LC-3's case)
PC and PCMUX: inputs: from bus, from PC++, from the result of Address Adder
(TRAP) (normal) (JMP, BR)
(controlled by PCMUX)
output: into bus

Assembly Programming

Tracing the Program

Single-stepping: one instruction at one time.

Breakpoints: tell the simulator to stop executing when reaching the specific instruction.

Watchpoints: tell the simulator to stop executing when a register or a memory location changes or reach a specific value.

Assembler

Assembler turns symbols into machine instructions.

Assembly Language (LC-3)

Three kinds of lines:

Comments: lines start with ";".
Instructions –
Directive – for the assemblers/software (also called "pseudo-op") e.g. #include<> (in C++)

Instruction Format: `LABEL OPCODE OPERANDS ; COMMENTS`
(optional, used in JMP/BR) (the "name" of the instruction) (store the address of the instruction)

Opcodes: *omitted*
Immediate numbers: `#[NUM]`: decimal numbers; `x[NUM]` : hexadecimal numbers
Separated by comma(",")

Directive Format: `LABEL OPCODE OPERANDS ; COMMENTS`

`.ORIG [Address]`	: set the starting address of program.
`.END`	: end of the program
`.BLKW [Number]`	: allocate [Number] words of storage //Block
`.FILL [Number]`	: allocate one word, initialize with [Number]
`.STRINGZ [Str]`	: allocate length(Str)+1 locations, initialize with [Str]+[null]

Trap Codes

`HALT`	`TRAP x25`	Halt all execution. Also, print message to console.
`IN`	`TRAP x23`	Read (and echo, i.e. print it in the console) one character from keyboard and store it in R0[7:0].
`OUT`	`TRAP x21`	Write one character in R0[7:0] to console.
`GETC`	`TRAP x20`	Read one character from keyboard and store it in R0[7:0].
`PUTS`	`TRAP x22`	Write null-terminated string to console. Address of string is in R0.

“`HALT`” and “`.END`”:

`HALT` will actually shut down the execution. Might also occur in branch.

Meanwhile, we need to console lines after `HALT` (value assignment, etc.). Thus, need “`.END`”.

Assembly Process

$$\begin{tikzcd}
\text{Assembly Language Program} \arrow[r] & \text{1st Pass} \arrow[rr] \arrow[dr] & & \text{2nd Pass} \arrow[r] & \text{Object File} \\
& & \text{Symbol Table} \arrow[ur] & &
\end{tikzcd}$$

1st Pass: Constructing Symbol Table

1) Find `.ORIG`. Get the address of the 1st instruction. Initialize LC (Location Counter) with it.

2) Find the next non-empty line.

    If label exists, add label and LC to Symbol Table.

    Change LC. (increment by 1 or the number of words allocated when `.BLKW`/`.STRINGZ`)

2nd Pass: Generate corresponding machine language instruction for each executable assembly language statement.

Potential Prob.s:

1) Improper number (Immediate number vs. Max bits for IMM)

2) Improper format

3) Address (associated with label) more than 256 from instruction – can’t use PC-relative Addressing Mode in this case!

Object File Format

The first line is not an instruction. Indicate the starting address.

How is a program loaded into MEM?

Loader

[.obj] → assembler → [.exe/.bin] → loader → scan the file and load instruction to required address. (the “`.ORIG`” directive’s demand)

e.g. `.ORIG x3000` → load the 1st instruction into x3000.

Linker

.obj can have external variables, etc. in other files.

Linker link several .obj files together (can see symbol table of all .obj files) to form an executable file (.exe, .bin, etc.).

I/O

I/O Controller

CPU ⟷ Control ↘

↘ Output Data → <u>Electronic</u> → Output Device. (display)

(perform actual operations)

↑ <u>Control and Output Data are two registers.</u>

Programming Interface

1) How to distinguish I/O with MEM?

① I/O Instructions

[I/O OPCODE] [DEVICE] [OPERATION (read/write)]

e.g. In LC-3, reserved opcode 1101 can be used as I/O Instruction.

<u>Problem:</u> We need to apply many changes to the CPU. (the state diagram)

② Memory-mapped

Assign a memory address to each device register. Use LD/ST to control and data transfer

2) Transfer Timing: Synchronous and Asynchronous

Synchronous – predictable rate -- CPU read/write every determined cycles

Asynchronous – unpredictable (e.g. keyboard and mouse)

-- CPU need to synchronous（同步）with the device →3)

3) Transfer Control

Polling: CPU Actively "ask"/check status register.

Interrupting: If have something, send message to CPU.

POLLING (CPU 轮询) -- instructions through "software"

In LC-3:

INPUT DEVICES:

A character is typed:

0) the input stored at KBDR (for ASCII, KBDR[7:0]).

1) Ready bit (KBSR[15]) is set to 1.

2) Keyboard is ignored. (while KBSR[15] is 1).

CPU continuously check the KBSR[15].

1) ready bit is "1", read, reset KBSR[15] to 0. – Can use BRzp to check "ready bit".

OUTPUT DEVICES:

Ready bit, DDR, DSR, …

INTERRUPT (中断机制) -- directly through hard-wire, through "hardware"

Priority. Can interrupt inferior tasks at hand.

In fact ready bit can be used as "interrupt enable". But we might need to program IE, thus we use KBSR[14] as IE. The true Interrupt signal is the AND of IE and ready bit.

LC-3: 8 Priorities (PL0~PL7).

HOW CPU check if there is an interrupt signal?

FETCH → DECODE → EVALUATE ADDRESS → FETCH OPERAND → EXECUTE OPERATION → STORE RESULT → <INTERRUPT SIGNAL?>

Y ↓ N → FETCH

JMP

→ ISR

LC-3 TRAP Mechanism

256 service routines. – part of the operating system. (Convention is that system code is stored below x3000)

Need: Where to go? Where to get back?

Stored at (0x0000 – 0x00FF), known as "System Control Block".

```
TRAP [1 1 1 1] [ 0 0 0 0 ] [ trap vector (8 bits) ]
```

1) Save old PC at R7. Meanwhile, ZEXT the trap vector.

2) ZEXTed TRAP VEC → MAR.

3) MDR ← MEM[MAR]

4) PC ← MDR

```
RET (JMP R7)
```

Return to where we came from.

PC: ←[TRAP]— TrapVec

Execute Service Routine.

PC ←[RET]— R7.

Continue the sequential instructions.

Then what if we need to use R7?

Saving and Restoring Registers

```
        .ORIG x0430          ; syscall address
        ST     R7, SaveR7
        ST     R1, SaveR1    ; In fact there's no need to backup R1,
but we don't know if R1 is used.
    Ck  LDI    R1, CRTSR     ; get status, check
        BRzp   Wt
    Wt  STI    R0, CRTDR     ; Write out
    Rt  LD     R1, SaveR1
```

```
LD     R7, SaveR7
CRTSR  .FILL  xF3FC
CRTDR  .FILL  xF3FF
SaveR1 .FILL  0
SaveR7 .FILL  0
.END
```

Callee-save: Before start the process, save registers' values that might be altered.

"Inside the Service Routine". Impossible for Service Routines with return values.

Caller-save: Before we calling a service routine, we backup the value of register that might be altered.

"Outside the Service Routine". A must for calling service routines with return values.

What if a callee calls another service routine?

– A problem is the backup of registers when you try to apply recursion to service routine.

User Code (Subroutine) – "functions"

Jump to Subroutine: `JSR [ 0 1 0 0 ] [ 1 ] [ PCoffset11 ]`

Similar to JMP and TRAP. 1) R7←PC. 2) PC←PC+SEXT[Offset]

`JSRR [ 0 1 0 0 ] [ 0 ] [00][ BASE ][000000]`

1) R7←PC. 2) PC←BASE

At the end of a subroutine, also use "RET" to get back to sequential instructions.

Interrupt and Stack

Stack: Often store [top] in R6.
POP/PUSH as subroutines → use JSR instruction to jump to the subroutines

Interrupt: Store the current State, jump to service (sub)routine.
Processor State?

- PC (Processor Counter)
- R0~R7
- Processor Status Register: n/z/p (condition codes), Privilege (P), Priority Level (PL)
  [0:2] [15] [10:8]

Supervisor Stack
Saved.SSP: register. Supervisor Stack Ptr. (SSP)
Saved.USP: register. User Stack Ptr. (USP)
Switching from user stack to supervisor stack, Saved.USP⟵R6. R6⟵Saved.SSP
Push PSR and PC into Supervisor Stack

How to return? -- RTI

Exception: something unexpected happens inside the processor → cause an exception
e.g. privileged operation (RTI in user mode), Executing an illegal opcode, divided by 0, accessing an illegal address

Handled like interrupts

ASCII to Binary: Lookup Table: 空间换时间

```
Lookup10    .FILL 10
            .FILL 20
            .FILL 30
            …
```

获得十位，视为索引，直接加上

C Programming

High-level Language

Symbolic names to values – no need to know which register

Abstraction of hardware – not limited by instructions e.g. c = a*b

Expressiveness +Readability e.g. if-else structure

Compilation vs Interpretation

Interpretation: one line at a time

$\rightarrow$ a program executing the program, need to install interpreter to execute (no output file)

Compilation: analyze the entire program, reduce possible operations

$\rightarrow$ generate executable program directly, no need to install compiler to execute (output file: .exe)

e.g.

```
get w from KBD
x = w + w
y = x + x
z = y + y
```

Interpreter: 3 operations

Compiler: (Mild compilers) z = 8*w

(Radical compilers, e.g. gcc – O2) z = w << 3

C Compilation

Preprocessor

macro substitution + conditional compilation

"source-level" transformations $\rightarrow$ output: still a C program

Compiler

generates object file

Linker

linking several obj files

Compiler

Source Code Analysis ("Front End") + Code Generation ("Back End") + Symbol Table

(Machine Code $\searrow$)

(ISA-independent) (ISA-dependent)

Can reuse Front End on different ISAs $\rightarrow$ can be transplanted

Preprocessor

```
#include <stdio.h>          /* -- copy the content of "stdio.h" into the source code
#define pi 3.14        /* a macro（宏）
```

-- scan the entire code and replace "pi" with "3.14". (in preprocessor)

-- after preprocess, all lines with "#define" is deleted. */

Variable Declarations/ IO

Operation Precedence（优先级）, Association(l-to-r/r-to-l)（左结合性/右结合性）

*,/,%____6; +,-_____5.

Class Conversion: mixed types, smaller type "promoted" to larger. (e.g. int+double→double)

Bitwise Operators:

~(bitwise not)	4
>>,<<	8
&(bitwise and)	11
^(bitwise xor)	12
|(bitwise or)	13
!(logical not)	4
&&(logical and)	14
||(logical or)	15
x++	2
x--	2
++x	3
--x	3

Symbol Table

Compiler-kept information: Name, Type, Offset(address), Scope (e.g. global, main, function, etc.)

Local variables stored in **activation record / stack frame.**

In LC-3, use R5 to mark the starting address of all local variables. (Reprise: R6: stack ptr)

Offset: negative. Since stack storage is from larger address to smaller address

Global variables stored in a fixed region called **Global Data Section**, just besides Instructions.

In LC-3, use R4 to mark the starting (smallest) address of Global Data Section.

Offset: positive. R4 is the smallest address.

The visit of MEM is time-consuming. Some compilers will recognize variables accessed frequently and allocate them in registers.

Control Structure

Conditional (If/Switch) /Iteration (While/For)

Functions – readability

- zero or multiple arguments passed in
- single result returned
- return value: a particular type

Declaration & Definition

Functions' implementation in ASM

1) Use register to store arguments and return value: Use R0. – limited value and 2number.

2) Use activation record. (storing information e.g. arguments and local variables)

Create a new A.R.. Push. Copy value into arguments. Call function.
    Execute code. Local variables also stored in A.R..
Put result in A.R.. Pop.

The starting address of A.R. is R5 (Frame Ptr). Stack top ptr is R6.
[[local variables | dynamic link | return address | return value] [arguments]]   : Activation Record
(the R5 before) (the R7 before) ← PC!
R6   R5 ____________ bookkeeping ____________
small –   – large
Offset: positive – arguments; non-positive – local variables

CALLER: R6++, PUSH arguments into run-time stack. Call R6.
CALLEE: allocate return value (R6++). Return Address←R7 (R6++), Dynamic Link←R5 (R6++).
    R5←@Dyn Link +1.
    Execute function code. Store result into return value.
    POP local variables.
    R5←Dynamic Link.(POP).
    R7←Return Address(POP).
    RET. (JMP R7)
CALLER: loads return value. POP arguments.

Pointers and Arrays
Why need ptrs to get address? – indirectly access data
    e.g. local variables' swap are meaningless in some cases. – 值参 vs 实参(&x)
Array: arrays with known length – in stack

Structs
存放方式与普通变量一致
访问方式类似数组（连续地址，偏移量静态已知）

Malloc + Free (Dynamic Allocation)
Function Prototype:
void *malloc(int numBytes)
returns a genetic pointer (void*), **i.e.** the starting address of a contiguous region of MEM of the requested size.

Implementation:
var = (Type*) malloc(sizeof(Type)*NeededNumber)
强制类型转换   （只能一个个 Byte 开辟空间）

void free(void*)
frees the dynamically allocated variables passed in through arguments

```
[ TRAPvec ][ Sys Subroutine ]… [ Program ][ global section ][ HEAP ][ Runtime stack]
                                            R4→               R3      ←R6
```

HEAP can use: R3~R6.

Microarchitecture

To improve performances.

- Concurrency
  - Von-Neumann vs. Harvard
  - Multiple instruction issue; Pipelining; Branch prediction
- Locality
  - Caches, Registers
- Clock rate

To improve power.

Introduction: Microarchitecture

– gate-level. generally use some high-level structures (e.g. ALU, Register, SRAM, DRAM)

(faster visit) (slower visit, MEM)

microarchitecture can change very fast while maintaining the original architecture.

Amdahl's Law

$$speedup_{overall} = \frac{executiontime_{old}}{executiontime_{new}} = \frac{1}{1 - Fraction_{enhanced} + \frac{Fraction_{enhanced}}{speedup_{enhanced}}}$$

i.e.

$$executiontime_{old} = 1 - \lambda + \lambda$$

$$executiontime_{new} = 1 - \lambda + \frac{\lambda}{N}$$

Microcode (ucode, μcode)

-- easier than connecting wires

-- separate data path and ctrl – reusable

e.g. ADD DR, SR1, SR2

corresponding microcode:

Regs[SR1]→ALU_RegA, Regs[SR2]→ALU_RegB, $\underline{\text{ADD}_{(ctrl)}\text{→ALU, ALU→Reg[DR],}}$

Fetch and decode next instruction [3 micro ops]

6 micro ops in total.

ADD DR, SR1, SR2, SR3?

– two ADD instructions (two instructions, another register needed, slower) – 12 micro ops

corresponding microcode:

Regs[SR1]→ALU_RegA, Regs[SR2]→ALU_RegB, $\underline{\text{ADD}_{(ctrl)}\text{→ALU, ALU→ALU\_RegA,}}$

Regs[SR3]→ALU_RegB, $\underline{\text{ADD}_{(ctrl)}\text{→ALU, ALU→Reg[DR],}}$ [5 micro ops]

Fetch and decode next instruction [3 micro ops]

8 micro ops in total.

speedup = 12/8 = 1.5

CISC: Complex Instruction Set Architecture: the length of an instruction is not determined
RISC: the length of an instruction is static (determined). e.g. LC-3

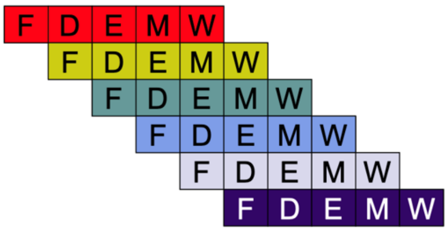
datapath, ctrl, next[condition?Y], next[condition?N] → HW6

Pipelining

Optimization Target: CPI (Clock Cycles per Instruction) or IPC (Instructions per Clock Cycle)

1) two Single-port register file: 1 Clock Cycle, complete "Regs[SR1]→A, Regs[SR2]→B" if A, B are not in the same register file.
2) Multi-port register file. 1 Clock Cycle, complete "Regs[SR1]→A, Regs[SR2]→B".

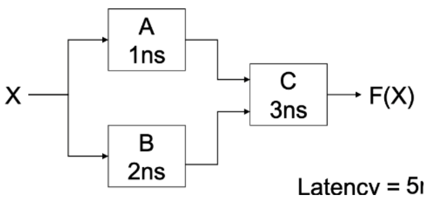
Pipelined processors



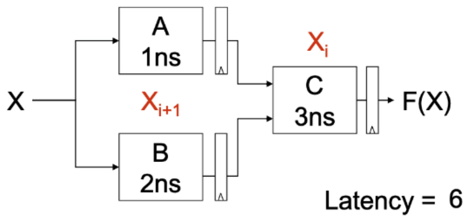
what if different part take different time?
an instruction → [2][4][6][2] ~ [6]: Bottleneck
unpipelined: 14
pipelined: 3*6+2=20 (NEED TO WAIT!) ← CPI↑ vs IPC↑

Latency（延迟）: time it takes to complete one instruction (20 in the case above).
Throughput（吞吐量）: #(computations done per unit time) (1/6 in the case above) = 1/（多少时间出一个结果）

Add registers to hold the temporary result in the process.

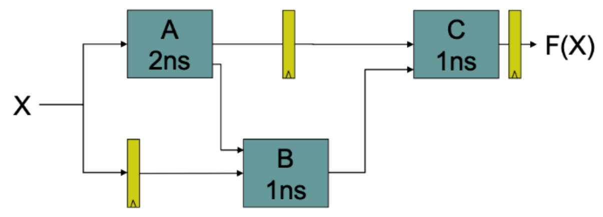


unpipelined: 5ns Latency, 1/5 Throughput



pipelining: 6ns Latency(3+3), 1/3 Throughput

↑ Better Throughput, Worse Latency



How many stage pipeline is this?
What is the problem?

4ns   Not optimized at all

Interleaving: for stages with larger latency, we can put several cells in the same stage.
e.g. 6 ns → three cells → can produce a result in every 2ns → Increase throughput

一般流水线化的方法：转化为有向无环图（DAG），输出端开始切割，一次切割覆盖所有输入到输出的路径。**切割处：寄存器**。

From the input to the output, the number of registers on every data path should be the same, otherwise is called **Malformed Pipeline.** May not work.（因为所有寄存器接的 CLK 是同一个）

Pipelined processor: Dependencies (or Hazards)

Data Dependency: the result would be used as inputs
Ctrl Dependency: branches (condition code), traps
Structural Dependency:

Solutions for data dependency:

1) Interlock 互锁

DECODE stage: record all destination registers of all instructions in process. (a new DST register enqueue, one DST register dequeue)

if the source register is one of the DST registers

– pause/stalled/operating a NOP op（FETCH 和 DECODE 阶段 stall 等待，重复当前 instruction；后面阶段重复 NOP 指令）until the DST register is written back (and is dequeued).
those operating NOP: pipeline bubble

2) Bypass

The result will be calculated at EXECUTION.

i) When operand is in register:

Therefore, we can "ignore" the current DECODE+FETCH OPERAND stage, directly fetch the result from the EXECUTION (and overwrite the original (and faulty) data fetched in the DEOCDE+FETCH OPERAND stage) at this time (EX of the previous instruction).

ii) When operand is in MEM:

stall + bypass

Data Dependencies

READ – R, WRITE – W, AFTER – A

RAR: No dependencies. **e.g.** R2 = R1+1, R3 = R1+3

WAW: No dependencies.

WAR: No dependencies. (1-EX before 2-WB)

RAW: True dependency.

Control Dependencies

```
e.g. I1 I2(BRnzp) I3 … I4(Brnzp dst.)
I1 [IF] [ID] [EX] [MA] [WB]
I2       [IF] [ID] [EX] [MA] [WB]
I3            [IF] [nop][nop]
I4                 [IF]
                    ↑ We add an adder to directly get PC after the BR instruction.
                      Also add a MUX in IF stage. (PC+1 or PC+Offset)
If not, we need to stall until EX is done.
```

```
e.g. I1 I2(BRn) I3 … I4(Brn dst.),    where I1: ADD/AND/NOT
I1 [IF] [ID] [EX] [MA] [WB]                    (in fact, n/z/p will change in WB stage)
I2       [IF] [ID] [EX] [MA] [WB]              (but we can know what n/z/p will be in EX)
I3             [IF] [ID] [EX] [MA] [WB]   (if condition not reached, not jumped)
I3             [IF] [nop][nop]
I4                 [IF]
```

```
e.g. I1, I2(BRn), I3, I5, … I4(BRn dst.),   where I1: LD/LDI/LDR
I1 [IF] [ID] [EX] [MA] [WB]                    (in fact, n/z/p will change in WB stage)
I2       [IF] [ID] [EX] [MA] [WB]              (but we can know what n/z/p will be in EX)
I3             [IF] [ID] [EX] [MA] [WB]   (if condition not reached, not jumped)
I3             [IF] [ID][nop] [nop]
I5                  [IF] [nop][nop]
I4                        [IF]
```

In fact, we predict the branch instruction won't cause a "jump". If not, we erase (or kill) instructions misfunctioned (e.g. I3, I5). Waste a pipeline slot.

Better Solution:

*** Branch Prediction** (not covered in this course)

*** Delay Slot: --** **change the semantic（延迟槽语义）**

Analyze the codes, find an instruction independent from the branch. Put it behind the branch instr.

No need to kill the instruction! （本来就需要做这一条指令，无论 branch 是否跳转，branch 后这条指令在原本代码中都需要被执行完毕）。

1 delay shot: (for ADD/AND/NOT)

- Original code

- 98: R2 = R4 + R5
- 100: R0 = R1 - R2
- 102: R3 = R4 * R5
- 104: BRn 200
- 306: R0 = R3 & R4

- 1 Delay slot

- 98: R2 = R4 + R5
- 100: R3 = R4 * R5
- 102: BRn 202
- 104: R0 = R1 - R2
- 306: R0 = R3 & R4

2 delay shots: (for LD/LDI/LDR)

- Original code

- 98: R2 = R4 + R5
- 100: R0 = R1 - R2
- 102: R3 = R4 * R5
- 104: BRn 200
- 306: R0 = R3 & R4

- 2 Delay slots

- 98: R3 = R4 * R5
- 100: BRn 204
- 102: R2 = R4 + R5
- 104: R0 = R1 - R2
- 306: R0 = R3 & R4

if there are no instructions can be moved into the delay slot, we add a NOP instruction.

Structural Dependencies: （硬件结构竞争）

If there is only one MEM: IF and MA cannot happen at the same time. (both will visit MEM)

e.g.

I1 [IF] [ID] [EX] [MA] [WB]

I2 [IF] [ID] [EX] [MA] [WB]

I3 [IF] [ID] [EX] [MA] [WB]

I4 [IF] [ID] …

Throughput ↓↓

Solution: Cache

Feedback: Used to resolve dependencies

generate signals for

` Stall

` Killing Instructions

` Controlling the Input of MUX (e.g. PC⟵{PC+1, PC+Offset})

Multiple Functional Units

IF → ID/Reg → SCIU/BPU/FPU/MCIU/LSU(Load-Store Unit) → WB → ID/Reg

(Single Cycle Integer Unit)(Multiple-Cycle Integer Unit)

(Branch Processing Unit)(Floating-Point Unit)

Pipelining Increases Clock Rate

TotalTime = #(Instructions) * CPI * CycleTime

where CPI = Clock Cycles Per Instructions

CycleTime = Time / Clock Cycles ~ （处理器的主频）

↖ Determined by the max time consumption of all stages

One way to increase efficiency: 拆分组合逻辑

LIMITS: 1）拆到只剩一个门——但也可以 interleaving 交替工作

2）存储部件有频率极限

3）Power Wall（主频↑ Power Consumption↑↑）

A Bottleneck: The Visit of MEM (time consumption of MEM Visit >> Reg Visit)

Memory

Memory:

- Registers
- SRAM
- DRAM → 内存条
- Disk
- CAM (*)

Accessing Memory:

1) Access by Address: SRAM, DRAM, Disk, Regs

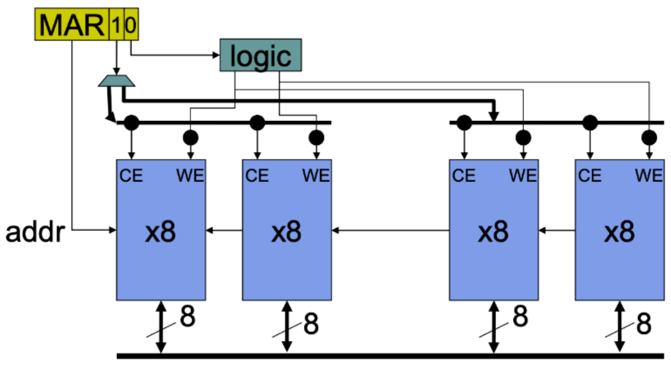
2) Access by Content: CAM (whether the thing we want is in it or not)

DRAM: can hold the data for only a second: need to be refreshed every particular time

only consist of a transistor (1T DRAM) – denser and cheaper

but hard to access

Multiple-Bank: 可以交替使用 DRAM → can visit MEM more frequently, **Interleaving**

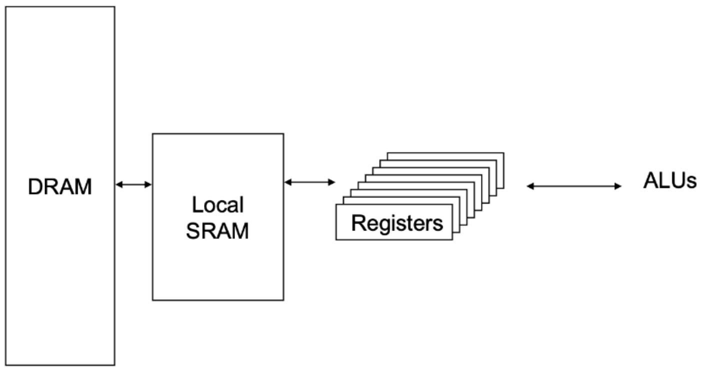


[0,4,8,12,…] [1,5,9,13,…] [2,6,10,14,…] [3,7,11,15,…]

Cache

Memory Wall – Processor 和 Memory 发展不平衡, Gap↑↑

Use SRAM to store data frequently accessed – Local SRAM



[Cache][$]

The visit of DRAM is super time-consuming.

Why can we do this? -- Temporal locality in Programs.

a cache line: hard-ware version of data structure -- (the address [tag] and its [value])

optimization:

1)* tag stored in <u>CAM</u> – whether the value we'll visit is in $ or not.

2) cache function. (like Hash)

the address $\rightarrow$ the address in [$]

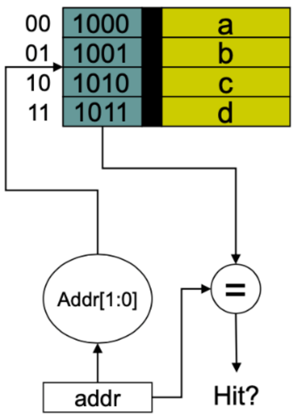
e.g. **Cache** [addressability: 2 bits] **MEM** [4 bits]

choose two bits of the address

just need to visit one cache line to see whether the thing we need is cache or not

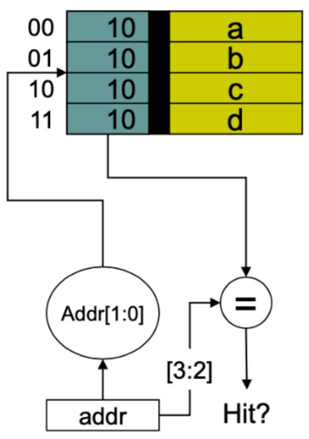
Direct-Mapped Cache

address cache with lowest order address bits



Yet another optimization: we don't need to store the full tag!

We just stored the highest order address bits (those not covered by the cache address)



Cache Hit Rate = #(Hits)/#(Accesses)

Cache Conflicts

Replaced.

The Difference of Hit Rate – Time Consumption: 100x!

e.g. 99% -- 0.99*1+0.01*100 = 1.99 cycles

97% -- 0.97*1+0.03*100 = 3.97 cycles

Cache Sets

0 [[tag value] [tag value]]

1 [[tag value] [tag value]] – cache set (one unit in a set is a cache line in this case, a $ set consists of 2 $ lines)

↑ 2-way set associative

In fact: In two SRAMs

e.g. 0 | | [100][1000's value] | | 0 | | [101] [1010's value] | |

0 | | [tag][value] | | 0 | | [tag][value] | |

1 | | [tag][value] | | 1 | | [tag][value] | |

SRAM0 SRAM1

Block

Enlarge the size of $ line – can store more than one value.

→ If we miss a data, we put the data itself and its following several address (as a block) into $.

e.g. 00 [tag] [value1] [value2] ~ the 2nd and 3rd lowest address (the lowest one: no need.)

01 [tag] [value1] [value2]

10 [1] [value1] [value2] ~ value1: address 1<u>10</u>0; value2: address 1<u>10</u>1; <u>1(tag)10(index)1(offset)</u>

11 [tag] [value1] [value2]

The size of a cache line determines the number of the offset.

The address in the MEM can be viewed as [tag | index (i bits) | offset (f bits)]
~ 2^i cache sets ~ 2^f units in a cache set

Thus, the size of the cache is

$$\begin{aligned} CacheSize &= (Offset \times UnitSize) \times CacheSets \times Ways \\ &= (CacheLineSize) \times CacheSets \times Ways \end{aligned}$$

(It's by definition.)

In fact, Validation Bit and Tag Bits are also a part of the cache line.

Locality

Misses:

Cold Misses (Compulsory Miss): First access.

Conflict Misses: Old data have to make room for new data. Make room for "conflicts". [Need more ways / associative blocks]

Capacity Misses: Not large enough. Have to make room.

Conflict – Replacement Policy

Direct Mapped: Only one. Replace it.

Associative Caches:

1) LRU (Least-Recently-Used)
2) Random
3) FIFO / LIFO (uncommon)

LRU

* bit map – when i-th block is referenced, set row i to 1 and column i to 0.

Write Back Policy

The value is changed by the program. When should we write it back?

Shall we update it simultaneously in the MEMORY? – time-consuming

Write-through: update simultaneously. Time-consuming

Write-back: only update when we need to replace its backup in the cache. (Write back on replacement.)

Optimized **Write-back:** add a bit showing whether the value is modified or not. (S-0/M-1)

MSI Protocol -- coherence

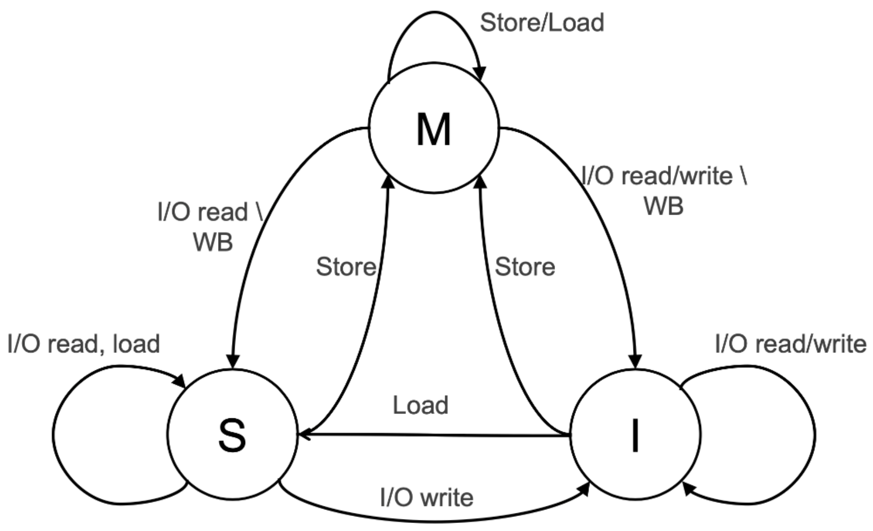
Cache coherence Protocol.

Three states: Modified, Shared, Invalid.

What if the value in the MEMORY changes while in cache remain the old state?

e.g. I/O（内存映射型输入输出设备） -- change the value in MEM!

Solution: <u>invalidate</u> the validation bit of the backup in the Cache.



Multi-layer Cache: Bigger Cache, Slower Speed.

CPU ⟷ L1 $ ⟷ L2 $ ⟷ MEMORY

(smaller) (larger)

Nowadays: 3 levels of caches

Instruction Cache and Data Cache: since each instruction requires Memory visit

Victim Cache: Group Associative Cache – 利用率不均.

Add a fully associative cache of a small size – 多出来的直接塞这里面

查询：并行查找组相联和全相联

Virtual Memory

-- used to: (1) avoid problems of system operating on physical memory of different size

small physical mem → large virtual address

(2) protection: physical memory only: entire memory is accessible for everyone

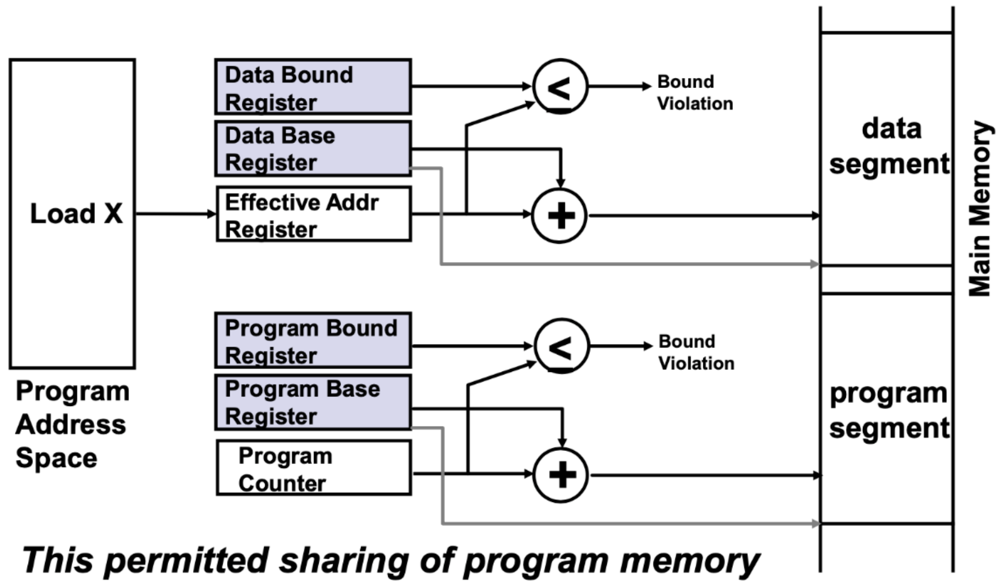
(3) different programs can run at the same time (multi-task)

Address Translation

symbolic name — virtual addresses → physical addresses (only processor knows)

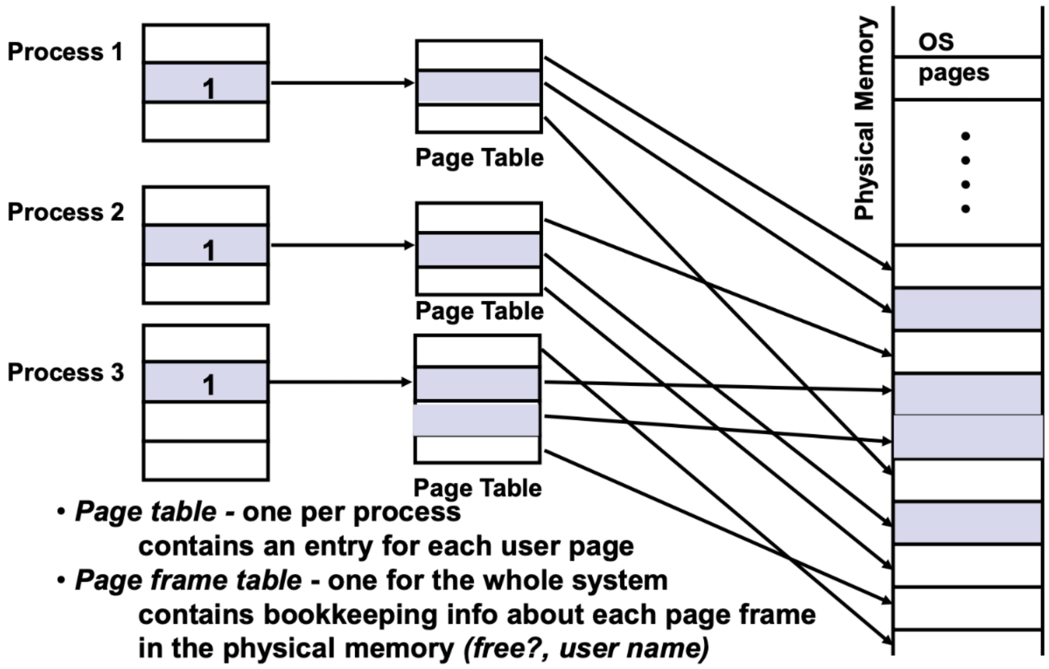
virtual address = Base + physical address

Single-Segment
Multi-Segment



contiguous address – MEMORY fragmentation

Solution:
Page-Based Translation
DISK ⟷ DRAM（视为 disk 的缓存, fully-associative） ⟷ Virtual Reality



• ***Page table*** **- one per process**
contains an entry for each user page
• ***Page frame table*** **- one for the whole system**
contains bookkeeping info about each page frame in the physical memory *(free?, user name)*

Page Table: #(virtual pages)×$\log_2$(bits of physical address)

Virtual Address: [Virtual Page Number] [Page Offset]

↓ mapping

Physical Address: [Physical Page Number] [Page Offset]　　　(also for cache: [tag] [index] [offset])

DISK ⟷ DRAM（视为 disk 的缓存, fully-associative）：一个个找耗时大 → Page Table!

<u>**Page Table:**</u> Recording Address Mapping. [Stored in PHYSICAL MEMORY]

access page table to get the physical address of the virtual address

a line in page table: Page Table Entry

[valid bit] [Access Rights] [Physical Page Number]

valid bit: 1(in DRAM); 0 (in DISC)

Access Rights: None, Read Only, Read/Write, Execute

one page table for one process/program

A "miss" is called a Page Fault（页缺失）.

Demanding Paging: <u>only load pages from disk when needed</u>