Contents

I Computer Abstractions and Technology

1. Introduction

1.1 History of Computers

1) Pre-computer: 计数的辅助工具
   a. The First Mechanical Calculating Machine
   b. Turing Machine: 计算与跳转
   c. First Electronic Digital Computing Device:
   d. First General-Purpose Electronic Computers
2) First Generation: Vacuum Tubes (真空管)
   a. von Neumann Architecture: 计算与储存分离，数据与指令保存在同一个储存器. 但遇到数据密集型此架构会有限制.

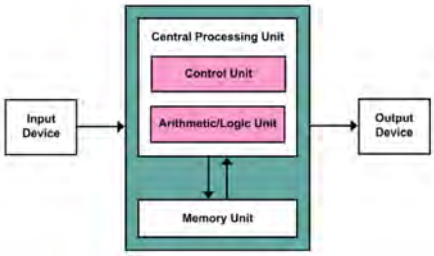


Figure 1: von Neumann Architecture

   b. Analog Computing:
   c. Digital Computing:
3) Second Generation: Transistors (晶体管)
   a. IBM
4) Third Generation: Integrated Circuits (集成电路)
5) Fourth Generation: Microprocessors
   a. The First Modern Personal Computer
   b. Commercial Personal Computer
   c. RISC Architecture
6) Fifth Generation:
   a. Quantum Computer
   b. Neuromorphic Computer

2. Computer Organization

1) 电子化的实现方式
2) 有指令集
3) 可执行指令
4) 可存储指令与数据
5) 计算能力上是图灵完全的

2.1 Decomposability of Computer Systems

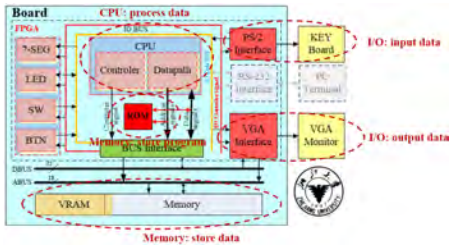


Figure 2: Five Classic Components of Hardware

Hardware:

- CPU
  - Control unit
  - Datapath
    - Path: multiplexors
    - ALU: adder, multiplier
    - Registers
    - ...
- Memory: 层级化
- I/O interface
  - Input: keyboard
  - Bidirectional: RS-232, USB
  - Output: VGA, LCD

Software:

- Application software (应用软件): word, PPT, office
- System software
  - Operation system (操作系统): Linux, macos
  - Compiler (编译器): GCC
  - Firmware (Driver software): 网卡驱动

指令集体系架构：连接硬件与软件 (抽象)

2.2 From a High-Level Language to the Language of Hardware

1) Machine language
2) Assembly language
3) High-level programming language

3. How to Build Processors

了解一下

3.1 Integrated Circuits (ICs)

主板核心

1) Manufacturing ICs

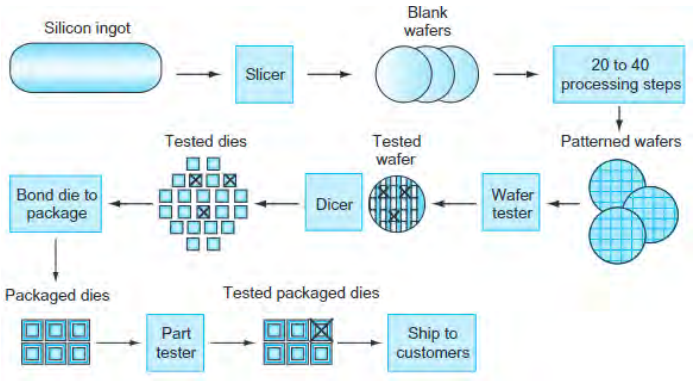


Figure 3: The chip maunfacturing process

2) Cost: Yield, proportion of working dies per wafer.

3.2 Challenges of the Processor

1) 内存墙
2) 功耗墙
3) 工艺发展限制

4. Computer Design:Performance and Idea

4.1 Response Time and Throughput

- Response/execution time: 响应时间 (one task)
- Throughput (bandwidth): 吞吐率 (total work)

4.2 Relative Performance

Define Performance=1/Execution

e.g. 10s on A, 15s on B, A is $\frac{\frac{1}{10}}{\frac{1}{15}} = 1.5$ times faster than B.

4.3 Measuring Execution Time

- Elapsed time: Total response time, including all aspects.
- CPU time: Time spent processing a given job.

$$\begin{aligned}\text{CPU Time} &= \text{CPU Clock Cycles} \times \text{Clock Cycle Time} \\ &= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}\end{aligned}$$

- Clock period: duration of a clock cycle.

e.g. $250\text{ps} = 0.25\text{ns} = 250 \times 10^{-12}\text{s}$

- Clock frequency: cycles per second.

e.g. $4.0\text{GHz} = 4000\text{MHz} = 4.0 \times 10^{9}\text{Hz}$

4.4 Instruction Count and CPI

$$\begin{aligned}\text{Clock Cycles} &= \text{Instruction Count} \times \text{Cycles per Instruction (CPI)} \\ \text{CPU Time} &= \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time} \\ &= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}\end{aligned}$$

Instruction Count for a program: Determined by program, ISA and compiler.

Average cycles per instruction: Determined by CPU hardware. If different instructions have different CPI, average CPI affected by instruction mix.

CPI in More Detail: If different instruction classes take different numbers of cycles

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

Weighted average CPI:

$$\begin{aligned}\text{CPI} &= \frac{\text{Clock Cycles}}{\text{Instruction Count}} \\ &= \sum_{i=1}^{n}\left(\text{CPI}_i \times \frac{\text{Instruction Count}_i}{\text{Instruction Count}}\right)\end{aligned}$$

$\frac{\text{Instruction Count}_i}{\text{Instruction Count}}$ is relative frequency.

4.5 Performance Summary

$$\text{CPU Time} = \frac{\text{Instruction}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

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

4.6 Power Trends

In CMOS IC technology

$$\text{Dynamic Power} = \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}$$

4.7 Multiprocessors

More than one processor per chip. Requires explicitly parallel programming.

4.8 Benchmark

1) SPEC CPU Benchmark: Programs used to measure performance, supposedly typical of actual workload. e.g. SPEC CPU2006.

2) SPEC Power Benchmark: Power consumption of server at different workload levels.

- Performance: ssj_ops/sec
- Power: Watts (Joules/sec)

4.9 Pitfall: Amdahl's Law

Improving an aspect of a computer and expecting a proportional improvement in overall performance.

$$T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvement factor}} + T_{\text{unaffected}}$$

Corollary: make the common case fast.

4.10 Pitfall: MIPS as a Performance Metric

MIPS: Millions of Instructions Per Second

$$\begin{aligned}\text{MIPS} &= \frac{\text{Instruction Count}}{\text{Execution Time} \times 10^6} \\ &= \frac{\text{Instruction Count}}{\frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}} \times 10^6} \\ &= \frac{\text{Clock Rate}}{\text{CPI} \times 10^6}\end{aligned}$$

5. Eight Great Ideas

1) Design for Moore's Law (设计紧跟摩尔定律)

2) Use Abstraction to Simplify Design (采用抽象简化设计)

3) Make the Common Case Fast (加速大概率事件)

4) Performance via Parallelism (通过并行提高性能)

5) Performance via Pipelining (通过流水线提高性能)

6) Performance via Prediction (通过预测提高性能)

7) Hierarchy of Memories (存储器层次)

8) Dependability via Redundancy (通过冗余提高可靠性)

II Arithmetic for Computer

1. Introduction

Computer words are composed of bits. Thus one word is a vector of binary numbers. There are 32 bit/word (word) or 64 bit/word (double word) in RISC-V. 32 bits contains four bytes.

Generic Implementation.

1) Use program counter (PC) to link to instruction address.

2) Fetch the instruction from memory.

3) The instruction tells what needs to be done.

4) ALU will perform the specified arithmetic operations.

2. Signed and Unsigned Numbers

2.1 Numbers and their representation

1) NUmber systems

$$(N)_k = \left(\sum_{i=m}^{n-l} b_i \cdot k^i \right)_k$$

- b: value of the digit.
- k: radix.
- n: digits left of radix point.
- m: digits right of radix point.

2) Representation

- ASCII - text characters (External)
- Binary number (Internal)

2.2 Number types

- Integer numbers, unsigned
- Signed numbers
  - Signed Number Representations: Sign Magnitud or Two's Complement
- Floating point number

3. Addition, subtraction and ALU

3.1 Addition & subtraction

- Adding bit by bit, carries → next digit.
- Subtraction
  - Directly.
  - Addition of 2's complement.

3.2 Overflow

Duoble sign-bits 判断溢出.

Table 1: General overflow conditions

Operation	Operand A	Operand B	Result
$A+B$	≥ 0	≥ 0	< 0
$A+B$	< 0	< 0	≥ 0
$A-B$	≥ 0	< 0	< 0
$A-B$	< 0	≥ 0	≥ 0

3.3 Constructing an ALU

是一个组合, 有多种计算方式, 使用了模块化的设计.

1) A full adder
2) Full adder Logic circuit
3) 1 bit ALU
4) Extended 1 bit ALU
   - Subtraction
   - Comparison
5) Complete ALU
6) Complete ALU — with Zero detector

3.4 ALU symbol & Control

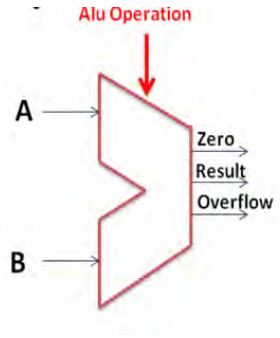


Figure 4: Symbol of the ALU

Table 2: Control: Function table

ALU Control Lines	Function
000	And
001	Or
010	Add
110	Sub
111	Set on less than
100	nor
101	srl
011	xor

nor 被去除了 (在 risc 里)
ALU Hardware Code ppt

3.5 More information

- Speed considerations
- Fast adders
- Carry Lookahead Adder (CLA)
- Addition formula in CLA
- Carry Lookahead Development
- Group Block Carry Lookahead
- Group Carry Lookahead Logic
- A plumbing analogy
- Carry skip adder
- Carry select adder (CSA)

4. Multiplication

4.1 Binary multiplication

Look at current bit position

Algorithm 1 Binary multiplication

if multiplier is 1 **then**
  add multiplicand
else
  add 0
end if
shift multiplicand left by 1 bit

4.2 Multiplier V1

1) **Logic Diagram:**
   - 64 bits: multiplier
   - 128 bits: multiplicand, product, ALU
2) **Algorithmic rule:** Requires 64 iterations
   - Addition
   - Shift
   - Comparison

Almost 200 cycles, very big, too slow!

4.3 Multiplier V2

Real addition is performed only with 64 bits. Least significant bits of the product don't change. New idea

- Don' t shift the multiplicand
- Instead, shift the product
- Shift the multiplier

ALU reduced to 64 bits.

1) **Logic Diagram:** Diagram of the V2 multiplier. Only left half of product register is changed.

2) **Algorithmic rule:** Addition performed only on left half of product register. Shift of product register.

4.4 Multiplier V3

Idea: use these lower 64 bits for the multiplier.

1) **Logic Diagram:**

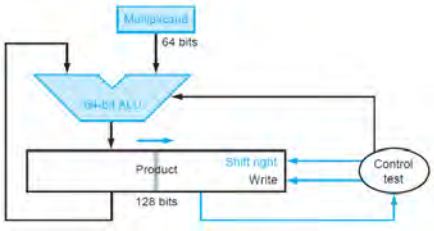


Figure 5: Multiplier V3

2) **Algorithmic rule:**

a. Set product register to '0'.

b. Load lower bits of product register with multiplier.

c. Test least significant bit of product register.

e.g. $Multiplicand \times Multiplier$: 0001×0111

Multiplicand:	**0001**	Shift out	
Multiplier:×	**0111**		
	00000111		#Initial value for the product
1	**00010111**		#After adding 0001, Multiplier=1
	00001011	**1**	#After shifting right the product one bit
	0001		
2	**00011011**		#After adding 0001, Multiplier=1
	00001101	**1**	#After shifting right the product one bit
	0001		#After adding 0001, Multiplier=1
3	**00011101**		
	00001110	**1**	#After shifting right the product one bit
	0000		
4	**00001110**		#After adding 0001, Multiplier=0
	00000111	**0**	#After shifting right the product one bit

Figure 6: Example for Multiplier V3

4.5 Signed multiplication

Basic approach:

1) Store the signs of the operands.

2) Convert signed numbers to unsigned numbers (most significant bit (MSB) = 0).

3) Perform multiplication.

4) If sign bits of operands are equal sign bit = 0, else sign bit = 1.

Improved method: Booth's Algorithm. Assumption: addition and subtraction are available.

4.6 Booth's Algorithm

Idea: If you have a sequence of '1's.

- subtract at first '1' in multiplier.
- shift for the sequence of '1's.
- add where prior step had last '1'.

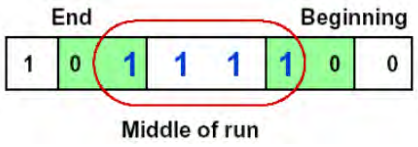


Figure 7: Idea

Result:

- Possibly less additions and more shifts.
- Faster, if shifts are faster than additions.

4.7 Booth's Algorithm rule

Action:

10 subtract multiplicand from left
11 no arithmetic operation-shift add 1
01 add multiplicand to left half
00 no arithmetic operation-shift add 0

Bit_{-1}='0'.

Arithmetic shift right:

- keeps the leftmost bit constant
- no change of sign bit

e.g.

- 2 * (-3) = - 6

0010 * 1101 = 1111 1010

- 13 * (-11) = - 143

01101 * 10101 = 11011 10001–>00100 01111

4.8 Faster Multiplication

Unrolls the loop.

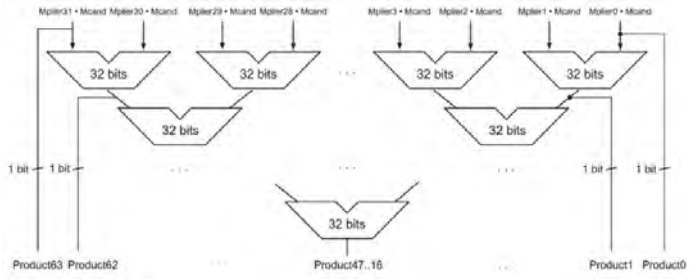


Figure 8: Faster Multiplication

5. Division

Algorithm 2 Division

Check for 0 divisor.

function LONG DIVISION APPROACH

if divisor ≤ dividend bits **then**

1 bit in quotient, subtract.

else

0 bit in quotient, bring down next dividend bit.

end if

end function

function RESTORING DIVISION

Do the subtract, and if remainder goes < 0, add divisor back.

end function

function SIGNED DIVISION

Divide using absolute values.

Adjust sign of quotient and remainder as required.

end function

5.1 Division V1

1) **Logic Diagram:** At first,the divisor is in the left half of the divisor register, the dividend is in the right half of the remainder register. Shift right the divisor register each step.

2) **Algorithmic rule:**

a. Subtract divisor

b. Depending on Result: Leaver or Restore.

c. Depending on Result: Write '1' or Write '0'.

5.2 Modified Division

1) **Logic Diagram:** Reduction of Divisor and ALU width by half. Shifting of the remainder. Saving 1 iteration. Remainder register keeps quotient. No quotient register required.

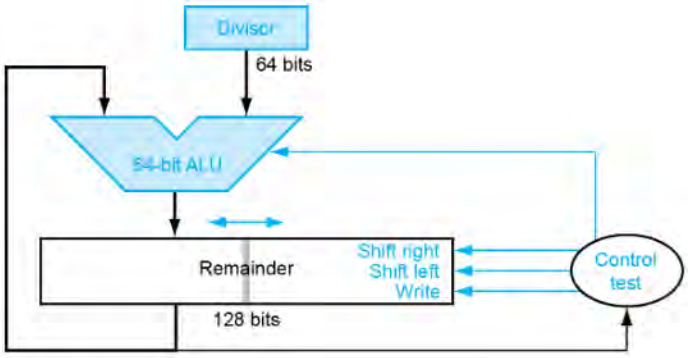


Figure 9: Modified Division

One cycle per partial-remainder subtraction. Looks a lot like a multiplier. Same hardware can be used for both.

2) **Algorithm:** Much the same than the last one. Except change of register usage.

e.g. 7/2 for Division V3, i.e. 0000 0111/0010

iteration	step	Divisor	Remainder
0	Initial Values	0010	0000 0111
	Shift Rem left 1	0010	**0000 1110**
1	1.Rem=Rem-Div	0010	1110 1110
	2b: Rem<0 →+Div,sll R,R_0=0	0010	**0001 1100**
2	1.Rem=Rem-Div	0010	1111 1100
	2b: Rem<0 →+Div,sll R,R_0=0	0010	**0011 1000**
3	1.Rem=Rem-Div	0010	0001 1000
	2a: Rem>0 →sll R,R_0=1	0010	**0011 0001**
4	1.Rem=Rem-Div	0010	0001 0001
	2a: Rem>0 →sll R,R_0=1	0010	**0010 0011**
	Shift left half of Rem right 1		**0001 0011**

Figure 10: Example for Division V3

5.3 Signed division

Keep the signs in mind for Dividend and Remainder. e.g.

- $(+7) \div (+2) = +3$, *Remainder* $= +1$

- $(-7) \div (+2) = -3$, $Remainder = -1$
- $(+7) \div (-2) = -3$, $Remainder = +1$
- $(-7) \div (-2) = +3$, $Remainder = -1$

One 64 bit register : Hi & Lo
Divide by 0 → overflow : Check by software

5.4 Faster Division

Can't use parallel hardware as in multiplier. Subtraction is conditional on sign of remainder.

Faster dividers (e.g. SRT division) generate multiple quotient bits per step. Still require multiple steps.

5.5 RISC-V Division

Four instructions:

- div, rem: signed divide, remainder
- divu, remu: unsigned divide, remainder

Overflow and division-by-zero don't produce errors. Just return defined results. Faster for the common case of no error

6. Floating point

Reasoning:

- Larger number range than integer rage
- Fractions
- Numbers like e and π

Representation:

- Sign
- Significand
- Exponent
- More bits for significand: more accuracy
- More bits for exponent: increases the range

Form: Normalised 3.634×10^{36}.
Binary notation: Normalised $1.xxxxx \times 2^{yyyyy}$.

6.1 Standardised format IEEE 754

- Single precision: 8 bit exp, 23 bit significand.
- Double precision: 11 bit exp, 52 bit significand.

Table 3: Single precision

31	30 23	22 0
S	exponent	fraction
1 bit	8 bit	23 bit

Table 4: Double precision

31	30 20	19 0
S	exponent	fraction
1 bit	11 bit	20 bit
31	fraction(continued)	0

Leading '1' bit of significand is implicit → saves one bit.

Exponent is biased:

- 00 ... 000 smallest exponent
- 11 ... 111 biggest exponent

Bias 127 for single precision. Bias 1023 for double precision.

Summary:

$$(-1)^{sign} \times (1 + significand) \times 2^{exponent-bias}$$

e.g.

6.2 Single-Precision Range

Exponents 00000000 and 11111111 reserved.

- **Smallest value**
  - Exponent: $00000001 \Rightarrow 1 - 127 = -126$
- **Largest value**

6.3 Double-Precision Range

6.4 Floating-Point Precision

Relative precision: all fraction bits are significant.

- Single: $\approx 2^{-23}$. Equivalent to $23 \times \log_1 02 \approx 23 \times 0.3 \approx 6$ decimal digits of precision.
- Double: $\approx 2^{-52}$. Equivalent to $52 \times \log_1 02 \approx 52 \times 0.3 \approx 16$ decimal digits of precision.

6.5 Limitations

- Overflow: The number is too big to be represented.
- Underflow: The number is too small to be represented.

6.6 Infinities and NaNs

- ±Infinity: Exponent=111 ... 1, Fraction=000 ... 0

Can be used in subsequent calculations, avoiding need for overflow check.

- Not-a-Number (NaN): Exponent=111 ... 1, Fraction$\neq$ 000 ... 0
Indicates illegal or undefined result. e.g. $\frac{0.0}{0.0}$.

6.7 Floating point addition

1) Alignment
2) The proper digits have to be added. Addition of significand
3) Normalisation of the result
4) Rounding

e.g. 0.5+(-0.4375) in binary

$$0.5_{10} = 1.000_2 \times 2^{-1}$$
$$-0.4375_2 = -1.110_2 \times 2^{-2}$$

1) The fraction with lesser exponent is shifted right until matches

$$-1.110_2 \times 2^{-2} \rightarrow -0.111_2 \times 2^{-1}$$

2) Add the significands

$$\begin{array}{r} 1.000_2 \times 2^{-1} \\ +) -0.111_2 \times 2^{-1} \\ \hline 0.001_2 \times 2^{-1} \end{array}$$

3) Normalize the sum and checking for overflow or underflow

$$0.001_2 \times 2^{-1} \rightarrow 1.000_2 \times 2^{-4}$$

4) Round the sum

$$1.000_2 \times 2^{-4} = 0.0625_{10}$$

Logic Diagram:

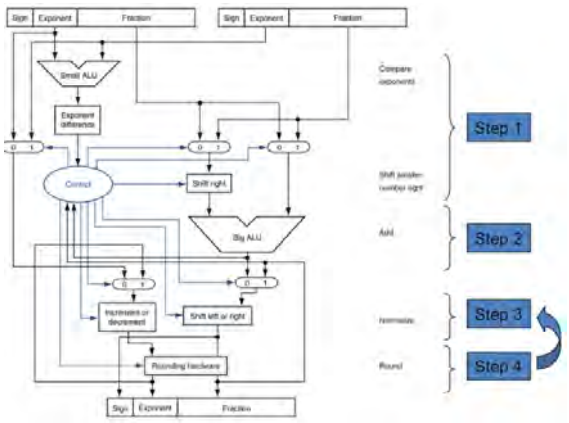


Figure 11: Logic Diagram

6.8 Multiplication

Composition of number from different parts → separate handling.

$$(s_1 \times 2^{e_1 - bias}) \times (s_2 \times 2^{e_2 - bias}) = (s_1 \times s_2) \times 2^{e_1 + e_2 - bias}$$

Logic Diagram:

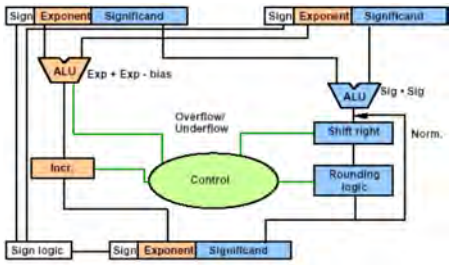


Figure 12: Logic Diagram

Algorithmic rule:

1) Add exponents
2) Multiply the significands
3) Normalise
4) Over- underflow
5) Rounding
6) Sign

6.9 Division — Brief

1) Subtraction of exponents
2) Division of the significants
3) Normalisation
4) Runding
5) Sign

6.10 Accurate Arithmetic

IEEE Std 754 specifies additional rounding control:

- Extra bits of precision (guard, round, sticky)
  - Guard: the first of two extra bits.
  - Round: method to make the immediate floating-point result fit the floating-point format.
- Choice of rounding modes
- Allows programmer to fine-tune numerical behavior of a computation.

Not all FP units implement all options. Not all FP units implement all options.

Trade-off between hardware complexity, performance, and market requirements.

Units in the last place(ulp): The number of bits in error in the least significant bits of the significant between the actual number and the number that can be represented.

III Instructions: Language of the Machine

The process of compiling:

1) High-level programming language
2) Assembly language
3) Machine language

1. Introduction

Language of the machine:

- Instructions -> words
- Instructions set -> vocabulary

Chosen: RISC-V

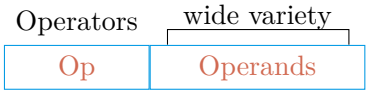
1.1 Instruction Set

CISC vs RISC

Table 5: CISC vs RISC

CISC complex instruction set computer	**RISC** reduce instruction set computer
Emphasis on hardware (>200 instructions)	Emphasis on software (<100 instructions)
Multiple instruction sizes and formats	Instructions of same set with few formats
Less registers	Uses more registers
More addressing modes	Fewer addressing modes (Load/Store)
Extensive use of microprogramming	Complexity in compiler
Instructions take a varying amount of cycle time	Instructions take one cycle time
Pipelining is difficult	Pipelining is easy

1.2 Instruction formats



Type of internal storage in processer. The number of the memory operand In the instruction.

1.3 Stored-program concept

Today's computers are built on 2 key principles (Stored-program concept):

1) Instruction are represented as numbers.
2) Programs can be stored in memory to be read or written just like numbers.

2. Arithmetic Operations

Every computer should perform arithmetic. Only one operation per instruction (Two sources and one destination). All arithmetic operations have this form.

Design Principle 1: Simplicity favors regularity. Regularity makes implementation simpler. Simplicity enables higher performance at lower cost

2.1 Arithmetic Example

C code

```
f=(g+h)-(i+j);
```

Compiled RISC-V code

```
add t0, g, h
add t1, i, j
sub f, t0, t1
```

3. Operands

Arithmetic instructions use register operands. RISC-V has a 32 × 64-bit register file. Use for frequently accessed data. 64-bit data is called a "doubleword". 32 x 64-bit general purpose registers x0 to x31. 32-bit data is called a "word".

Design Principle 2: Smaller is faster. c.f. main memory: millions of locations

3.1 RISC-V Registers

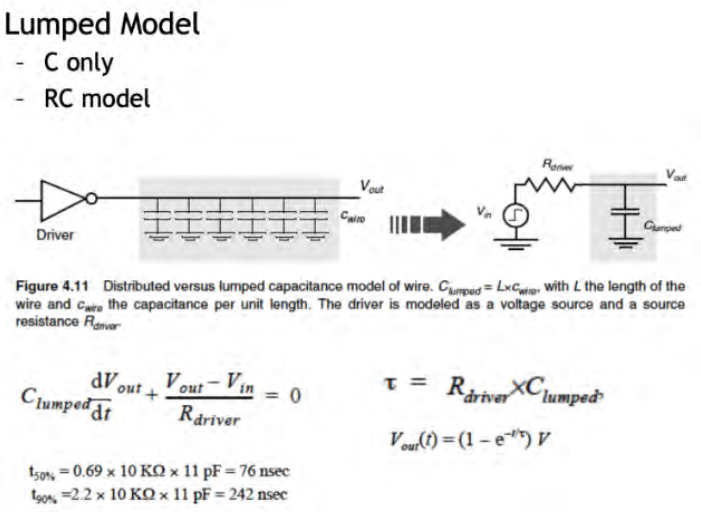


Figure 13: RISC-V Registers

Table 6: RISC-V Registers table

Name	Register Name	Usage	Preserved On Call?
x0	0	The constant value 0	n.a.
x1(ra)	1	Return address(link register)	yes
x2(sp)	2	Stack pointer	yes
x3(gp)	3	Global pointer	yes
x4(tp)	4	Thread pointer	yes
x5-x7	5-7	Temporaries	no
x8-x9	8-9	Saved	yes
x10-x17	10-17	Arguments/results	no
x18-x27	18-27	Saved	yes
x28-x31	28-31	Temporaries	no

3.2 Register Operand Example

C code

```
f = (g + h) - (i + j);
```

f, g, h, i, j in x19, x20, x21, x22, x23.

Compiled RISC-V code

```
add x5, x20, x21
add x6, x22, x23
sub x19, x5, x6
```

3.3 Memory Operands

Main memory used for composite data e.g. Arrays, structures, dynamic data. To apply arithmetic operations:

- Load values from memory into registers
- Store result from register to memory

Memory is byte addressed. Each address identifies an 8-bit byte.

RISC-V is Little Endian. Least-significant byte at least address of a word. c.f. Big Endian: most-significant byte at least address

RISC-V does not require words to be aligned in memory, unlike some other ISAs.

3.4 Memory Alignment

```
struct {
    int a;
    char b;
    char c[2];
    char d[3];
    float e;
```

```
}
```

e			
d[1]	d[2]	No use	No use
b	c[0]	c[1]	d[0]
a			

因为一次只能读出 4 字节内存中的一行, 如此 e 变量能一次读出.

3.5 *Endianness/byte order*

Big end: Leftmost e.g. PowerPC 01 02 = 258.
Little end: Rightmost e.g. RISC-V 01 02 = 513.

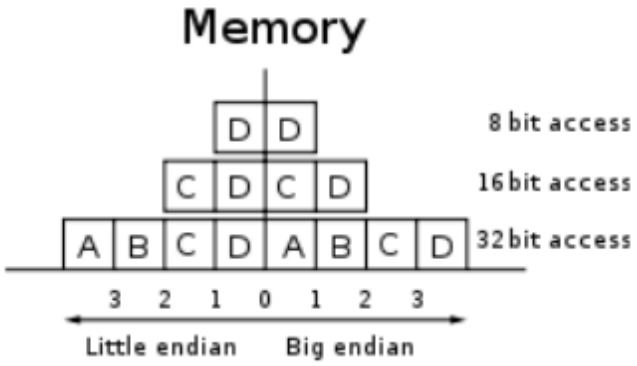


Figure 14: Endianness

3.6 *Memory Operand Example*

C code

```
A[12]=h+A[8]
```

h in x21, base address of A in x22.

1) Index 8 requires offset of 64, 8 bytes per double-word.

2) Offset: the constant in a data transfer instruction.

3) Base register: the register added to form the address.

Compiled RISC-V code

```
ld x9, 64(x22)
add x9, x21, x9
sd x9, 96(x22)
```

ld means load double word, lw means load word. Same sd means save double word.

3.7 *Register vs Memory*

Registers are faster to access than memory. Compiler must use registers for variables as much as possible. Only spill to memory for less frequently used variables.

3.8 *Constant or immediate Operands (立即数)*

Avoids the load instruction.

Compiled RISC-V code

```
addi x22, x22, 4 // x22=x22+4
```

Constant zero: a register x0.

Design Principle 3: Make common case fast.

3.9 *Brief summary*

Table 7: RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add x5,x6,x7	x5=x6 + x7	Add two source register operands
	subtract	sub x5,x6,x7	x5=x6 - x7	First source register subtracts second one
	add immediate	addi x5,x6,20	x5=x6+20	Used to add constants
Data transfer	load doubleword	ld x5, 40(x6)	x5=Memory[x6+40]	doubleword from memory to register
	store doubleword	sd x5, 40(x6)	Memory[x6+40]=x5	doubleword from register to memory

Table 8: RISC-V operands

Name	Example	Comments
32 registers	x0-x31	Fast locations for data. In RISC-V, data must be in registers to perform arithmetic. Register x0 always equals 0.
2^{61} memory words	Memory[0], Memory[8], ..., Memory[18,446,744,073,709,551,608]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential doubleword accesses differ by 8. Memory holds data structures, arrays, and spilled registers.

4. Signed and unsigned numbers

In RISC-V instruction set

- lb: sign-extend loaded byte
- lbu: zero-extend loaded byte

5. Representing Instructions

RISC-V instructions:

- Encoded as 32-bit instruction words
- Small number of formats encoding operation code (opcode), register numbers, ...
- Regularity

5.1 *Example: Translating Assembly Code*

Compiled RISC-V code

```
add x9, x20, x21
```

Decimal version of machine code

0	21	20	0	9	51

Binary version of machine code

0000000	10101	10100	000	01001	0110011
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

5.2 Hexadecimal

Base 16: Compact representation of bit strings, 4 btis pre hex digit.

e.g. eca8 6420: 1110 1100 1010 1000 0110 0100 0010 0000

可以打表记指令

5.3 RISC-V R-Format Instructions (R 型指令)

fnuct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Instruction fields:

- opcode: operation code
- rd: destination register number
- funct3: 3-bit function code (additional opcode)
- rs1: the first source register number
- rs2: the second source register number
- funct7: 7-bit function code (additional opcode)

All instructions in RISC-V have the same length.

5.4 RISC-V I-Format Instructions (I 型指令)

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

Immediate arithmetic and load instructions:

- rs1: source or base address register number
- immediate: constant operand, or offset added to base address, e.g. 2s-complement, sign extended

e.g. ld x9, 64(x22)

- 22 (x22) is placed rs1
- 64 is placed immediate
- 9 (x9) is placed rd

5.5 RISC-V S-Format Instructions (S 型指令)

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Different immediate format for store instructions

- rs1: base address register number
- rs2: source operand register number
- immediate: offset added to base address. Split so that rs1 and rs2 fields always in the same place.

e.g. sd x9, 64(x22)

- 22 (x22) is placed rs1
- 64 is placed immediate
- 9 (x9) is placed rs2

5.6 RISC-V instruction encoding

Table 9: RISC-V instruction encoding

Name	Format	Example						Comment
add	R	0	3	2	0	1	51	add x1, x2, x3
sub	R	32	3	2	0	1	51	sub x1, x2, x3
addi	I	1000		2	0	1	19	addi x1,x2,1000
ld	I	1000		2	3	1	3	ld x1, 1000(x2)
sd	S	63	1	2	3	8	35	sd x1, 1000(x2)

Two key principles of today's computers:

1) Instructions are represented as numbers (Stred-program)

2) Programs can be stored in memory like numbers

5.7 RISC-V fields (format)

Table 10: RISC-V fields (format)

Name	Fields						Comments
Field size	7bits	5bits	5bits	3bits	5bits	7bits	All RISC-V instruction 32 bits
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	*imm[12,10:5]*	rs2	rs1	funct3	*imm[4:1,11]*	opcode	Conditional branch format
UJ-type	*immediate[20,10:1,11,19:12]*				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

5.8 Stored Program Computer

Before, the program is stored as data. Future, the data may be the model.

6. Logical operations

Table 11: Instructions for bitwise manipulation

Operation	C	Java	RISC-V
Shift left	<<	<<	slli
Shift right	>>	>>	srli
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit XOR	^	^	xor, xori
Bit-by-bit NOT	~	~	

Useful for extracting and inserting groups of bits in a word.

6.1 Shift Operations

funct6	immed	rs1	funct3	rd	opcode
6 bits	6 bits	5 bits	3 bits	5 bits	7 bits

immed: how many positions to shift.

6.2 AND Operations

Useful to mask bits in a word. Select some bits, clear others to 0.

6.3 OR Operations

Useful to include bits in a word. Set some bits to 1, leave others unchanged.

6.4 XOR Operations

Differencing operation. Set some bits to 1, leave others unchanged.

Table 12: RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Logical	and	and x5, x6, 3	x5=x6 & 3	Arithmetic shift right by register
	inclusive or	or x5,x6,x7	x5=x6 | x7	Bit-by-bit OR
	exclusive or	xor x5,x6,x7	x5=x6 ^ x7	Bit-by-bit XOR
	and immediate	andi x5,x6,20	x5=x6 & 20	Bit-by-bit AND reg. with constant
	inclusive or immediate	ori x5,x6,20	x5=x6 | 20	Bit-by-bit OR reg. with constant
	exclusive or immediate	xori x5,x6,20	X5=x6 ^ 20	Bit-by-bit XOR reg. with constant
Shift	shift left logical	sll x5, x6, x7	x5=x6 << x7	Shift left by register
	shift right logical	srl x5, x6, x7	x5=x6 >> x7	Shift right by register
	shift right arithmetic	sra x5, x6, x7	x5=x6 >> x7	Arithmetic shift right by register
	shift left logical immediate	slli x5, x6, 3	x5=x6 << 3	Shift left by immediate

7. Instructions for making decision

Branch instructions: Branch to a labeled instruction if a condition is true. Otherwise, continue sequentially.

```
beq rs1, rs2, L1
```

If (rs1 == rs2) branch to instruction labeled L1

```
bne rs1, rs2, L1
```

If (rs1 != rs2) branch to instruction labeled L1

7.1 Example Compiling an if statement

Assume: f ∼ j — x19 ∼ x23

C code

```
if ( i == j ) goto L1 ;
f = g + h ;
L1: f = f - i ;
```

RISC-V assembly code

```
beq x21, x22, L1
add x19, x20, x21
L1: sub x19, x19, x22
```

1) go to L1 if i equals j
2) f = g + h (skipped if i equals j)
3) f = f - i (always executed)

7.2 Compiling if-then-else

Assume: f ∼ j — x19 ∼ x23

C code

```
if ( i == j ) f = g + h ;
else f = g - h ;
```

RISC-V assembly code

```
bne x22, x23, Else
add x19, x20, x21
beq x0, x0, EXIT
Else: sub x19, x20, x21
Exit: ..... statement
```

1) go to Else if i != j
2) f = g + h (Executed if i == j if)
3) go to Exit
4) f = g - h (Executed if i != j else)
5) the first instruction of the next C

7.3 *Compiling LOOPs*

Assume: g ∼ j — x19 ∼ x23 base of A[i] — x25

C code

```
Loop:
    g = g + A[i] ; // A is an array of 100
        words
    i = i + j ;
    if ( i != h ) goto Loop ;
```

RISC-V assembly code

```
Loop:
    slli x10, x22, 3
    add x10, x10, x25
    ld x19, 0(x10)
    add x20, x20, x19
    add x22, x22, x23
    bne x22, x21, Loop
```

1) temp reg x10 = 8 * i
2) x10 = address of A[i]
3) temp reg x19 = A[i]
4) g = g + A[i]
5) i = i + j
6) go to Loop if i != h

7.4 *Compiling while*

Assume: i ∼ k — x22 and x24 base of save — x25

C code

```
while ( save[i] == k )
    i = + i ;
```

RISCV assembly code

```
Loop:
    slli x10, x22, 3
    add x10, x10, x25
    ld x9, 0(x10)
    bne x9, x24, Exit
    addi x22, x22, 1
    beq x0, x0, Loop
Exit:
```

1) temp reg $t1 = 8 * i
2) x10 = address of save[i]
3) x9 gets save[i]
4) go to Exit if save[i] != k
5) i += 1
6) go to Loop

7.5 *More Conditional Operations*

```
blt rs1, rs2, L1
```

if (rs1 < rs2) branch to instruction labeled L1

```
bge rs1, rs2, L1
```

if (rs1 >= rs2) branch to instruction labeled L1

7.6 *Signed vs. Unsigned*

- Signed comparison: blt, bge.
- Unsigned comparison: bltu, bgeu.

7.7 *Hold out Case/Switch*

Used to select one of many alternatives. Compiling a switch using jump address table.

Assume: f ∼ k — x20 ∼ x25 x5 contains 4/8

C code

```
switch ( k ) {
    case 0 : f=i+j; break; /* k=0 */
    case 1 : f=g+h; break; /* k=1 */
    case 2 : f=g-h; break; /* k=2 */
    case 3 : f=i-j; break; /* k=3 */
}
```

RISC-V assembly code

```
blt x25, x0, Exit
bge x25, x5, Exit
slli x7, x25, 3
add x7, x7, x6
ld x7, 0(x7)
jalr x1, 0(x7)
```

1) test if k < 0
2) if k >= 4, go to Exit
3) temp reg x7 = 8 * k (0<=k<=3)
4) x7 = address of JumpTable[k]
5) temp reg x7 gets JumpTable[k]
6) jump based on register x7(entrance)

jump address table: x7=x6+8*k

Memory1

```
L0: address
L1: address
```

```
L2: address
L3: address
```

Memory2

```
L0:
    add $s0, $s3, $s4
    jalr x0, 0(x1)
L1:
    add $s0, $s1, $s2
    jalr x0, 0(x1)
L2:
    sub $s0, $s1, $s2
    jalr x0, 0(x1)
L3:
    sub $s0, $s3, $s4
    jalr x0, 0(x1)
```

- L0
  1) k = 0 so f gets i + j
  2) end of this case so go to Exit
- L1
  1) k = 1 so f gets g + h
  2) end of this case so go to Exit
- L2
  1) k = 2 so f gets g - h
  2) end of this case so go to Exit
- L3
  1) k = 3 so f gets i - j
  2) end of this case so go to Exit

7.8 Jump register & jump address table

Jump with register content

```
jalr x1, 100(x6)
```

Jump address table: x7 ← x6 + 4/8*K

7.9 Important conception — Basic Blocks

A basic block is a sequence of instructions with

- No embedded branches (except at end)
- No branch targets (except at beginning)

A compiler identifies basic blocks for optimization. An advanced processor can accelerate execution of basic blocks.

8. Supporting procedures

Procedure/function be used to structure programs. A stored subroutine that performs a specific task based on the parameters with which it is provided, easier to understand, allow code to be reused.

1) Place Parameters in a place where the procedure can access them
2) Transfer control to the procedure: jump to
3) Acquire the storage resources needed for the procedure
4) Perform the desired task
5) Place the result value in a place where the calling program can access it
6) Return control to the point of origin

8.1 Procedure Call Instructions

- Instruction for procedures: jal (jump-and-link)

Caller

```
jal x1, ProcedureAddress
```

1) Address of following instruction put in x1 (PC+4 → ra).
2) Jumps to target address.

- Procedure return: jalr (jump and link register)

Callee

```
jalr x0, 0(x1)
```

1) Like jal, but jumps to 0 + address in x1.
2) Use x0 as rd (x0 cannot be changed).

Can also be used for computed jumps.

8.2 Using More Registers

More Registers for procedure calling:

- a0 ∼ a7(x10-x17): eight argument registers to pass parameters & return values
- ra/x1: one return address register to return to origin point

Stack: ideal data structure for spilling registers. push, pop and Stack pointer (sp).

Stack grow from higher address to lower address.

- Push: sp= sp - 8
- Pop: sp = sp + 8

e.g. Compiling a leaf procedure (Assume: g, ..., j in x10, ..., x13 and f in x20)

C code

```
ll leaf(ll g, ll h, ll i, ll j){
    ll f;
    f = (g + h) - (i + j);
    return f;
}
```

RISC-V assembly code

```
addi sp, sp, -24
sd x5, 16(sp)
sd x6, 8(sp)
sd x20, 0(sp)

add x5, x10, x11
add x6, x12, x11
sub x20, x5, x6
addi x10, x20 ,0

ld x20, 0(sp)
ld x6, 8(sp)
ld x5, 16(sp)
addi sp, sp, +24
jalr x0, 0(x1)
```

- PUSH

  1) adjust stack to make room for 3 items.
  2) These three instructions save three register x5, x6 (Save value), x20 (Return value).

- LEAF

  1) register x5 contains g + h
  2) register x6 contains i + j
  3) f = x5- x6, which is (g + h) −(i + j)
  4) copy f to return register (x10 = x20 + 0)

- POP

  1) restore register x20 for caller
  2) restore register x6 for caller
  3) restore register x5 for caller
  4) adjust stack to delete 3 items
  5) jump back to calling routine

But maybe some of the three are not used by the caller. So, this way might be inefficient to save x5, x6, x20 on stack. Two classes of registers

- t0 ∼ t6: 7 temporary registers, by the callee not preserved
- s0 ∼ s11: 12 saved registers, must be preserved If used

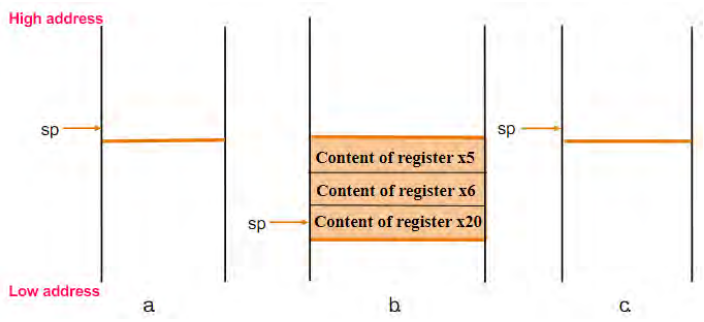


Figure 15: The values of the stack pointer and stack before, during and after procedure call

Conflict over the use of register both. Push all the registers to stack.

- Caller: pushes a0∼a7 or t0∼t6
- Callee: pushes ra (return address) and s0∼s11

8.3 Nested Procedures

Compiling a recursive procedure (Assume: n — a0)

C code for n!

```
int fact(int n){
    if(n<1)return 1;
    else return n*fact(n-1);
}
```

RISC-V assembly code

```
fact:
    addi sp, sp, -16
    sd ra, 8(sp)
    sd a0, 0(sp)
    addi t0, a0, -1
    bge t0, zero, L1
    addi a0, zero, 1
    addi sp, sp, 16
    jalr zero, 0(ra)
L1:
    addi a0, a0, -1
    jal ra, fact
    addi t1, a0, zero
    ld a0, 0(sp)
    ld ra, 8(sp)
    addi sp, sp, 16
    mul a0, a0, t1
```

```
18      jalr zero, 0(ra)
```

- fact
  1) adjust stack for 2 items
  2) save the return address: x1
  3) save the argument n: x10
  4) x5 = n - 1
  5) if n >= 1, go to L1(else)
  6) return 1 if n <1
  7) Recover sp
  8) return to caller
- L1
  1) n >= 1: argument gets (n - 1)
  2) call fact with (n - 1)
  3) move result of fact(n - 1) to x6(t1)
  4) return from jal: restore argument n
  5) restore the return address
  6) adjust stack pointer to pop 2 items
  7) return n*fact (n - 1)
  8) return to the caller

Preserved things across a procedure call.

- Saved registers(s0 ~ s11), stack pointer register($sp),
- return address register(ra/x1), stack **above** the stack pointer.

Not preserved things across a procedure call.

- Temporary registers(t0 ~ t7), argument registers(a0 ~ a7),
- return value registers(a0 ~ a7), stack **below** the stack pointer.

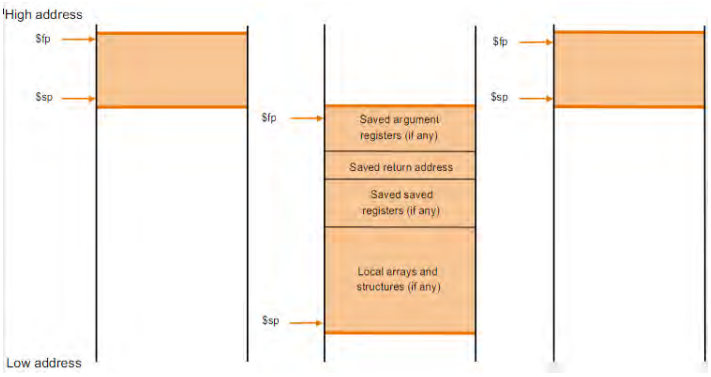


Figure 16: Stack allocation before, during and after procedure call

8.4 What is and what is not preserved across a procedure call

Table 13: across a procedure call

Preserved	Not preserved
Saved registers: x8-x9, x18-x27	Temporary registers: x5-x7, x28-x31
Stack pointer register: x2(sp)	Argument/result registers: x10-x17
Frame pointer: x8(fp)	
Return address: x1(ra)	
Stack above the stack pointer	Stack below the stack pointer

8.5 Memory Layout

- Text: program code
- Static data: global variables
- Dynamic data: heap
- Stack: automatic storage
- Storage class of C variables:
  - automatic
  - static

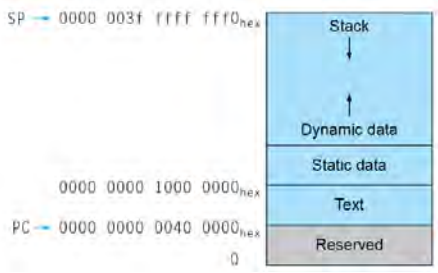


Figure 17: Memory Layout

8.6 Local Data on the Stack

Allocating Space for New Data on the Stack

- Procedure frame/activation record: The segment of stack containing a procedure's saved registers and local variables.
- Frame pointer
  1) A value denoting the location of saved register and local variables for a given procedure
  2) Local data allocated by callee

Used by some compilers to manage stack storage

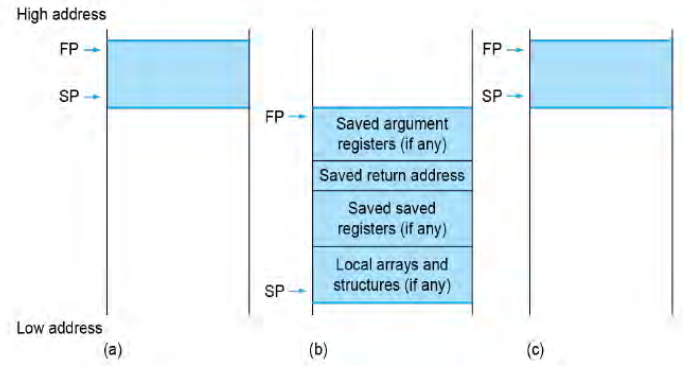


Figure 18: Local Data on the Stack

9. Communicating with Character Data

Byte-encoded character sets

- ASCII (American Standard Code for Information Interchange): 128 characters (95 graphic, 33 control)
- Latin-1: 256 characters (ASCII, +96 more graphic characters)

9.1 Byte/Halfword/Word Operations

RISC-V byte/halfword/word load/store

- Load byte/halfword/word: Sign extend to 64 bits in rd

```
lb rd, offset(rs1)
lh rd, offset(rs1)
lw rd, offset(rs1)
```

- Load byte/halfword/word unsigned: 0 extend to 64 bits in rd

```
lbu rd, offset(rs1)
lhu rd, offset(rs1)
lwu rd, offset(rs1)
```

- Store byte/halfword/word: Store rightmost 8/16 /32 bits

```
sb rs2, offset(rs1)
sh rs2, offset(rs1)
sw rs2, offset(rs1)
```

9.2 String Copy Example

Compiling a string copy procedure (Assume: base addresses for i – x19, x's base – x10, y's base – x11)

C code: Y → X

```
void strcpy(char x[], char y[]){
    size_t i;
    i=0;
    while((x[i]=y[i])!='\0')
        i+=1;/* copy and test byte */
}
```

RISC-V assembly code

```
strcpy:
    addi sp, sp, 8
    sd s3, 0(sp)
    addi s3, zero, zero
L1:
    add t0, s3, a1
    lbu t1, 0(t0)
    add t2, s3, a0
    sb t1, 0(t2)
    beq t1, zero L2
    addi s3, s3, 1
    jal zero, L1
L2:
    ld s3, 0(sp)
    addi sp, sp, 8
    jalr zero, 0(x1)
```

- strcpy
  1) adjust stack for 1 doubleword
  2) save x19
  3) i = 0
- L1
  1) x5 = address of y[i]
  2) x6 = y [i]
  3) x7 = address of x[i]
  4) x[i] = y[i]
  5) if y[i] == 0 then exit
  6) i = i + 1
  7) next iteration of loop
- L2
  1) restore saved old s3
  2) pop 1 double word from stack
  3) return

Optimization: Because strcpy is a leaf procedure, can allocate i to a temporary register s3/x18.

For a leaf procedure, the compiler exhausts all temporary register, then use the registers it must save.

10. Addressing for 32-Bit Immediate and Addresses

Wide Bit Immediate addressing:

- most constants is short and fit into 12-bit field
- Set upper 20 bits of a constants in a register with load upper immediate (lui rd, constant)

```
lui x19, 976
```

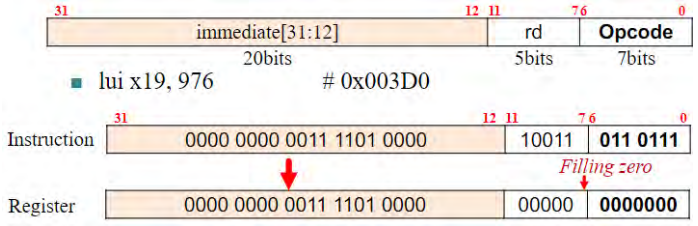


Figure 19: instruction format (U-type)

10.1 32-bit Constants

Loading a 32-bit constant

$$0000\ 0000\ 0011\ 1101\ 0000\ 1001\ 0000\ 0000$$
$$=(976 * 16^3 + 2304 = 4000000)_{10}$$

RISC-V code

```
lui s3, 976
addi s3, s3, 230
```

1) 976 decimal = 0000 0000 0011 1101 0000 binary (The value of s3 afterward is: 0000 0000 0011 1101 0000 0000 0000 0000)

2) 2304 decimal = 1001 0000 0000 binary

10.2 Branch Addressing

Addressing in branches:

- Branch instructions specify opcode, two registers, target address.
- Most branch targets are near branch, forward or backward.

SB-type: 2000 = 0111 1101 0000

```
bne x10, x11, 2000
```

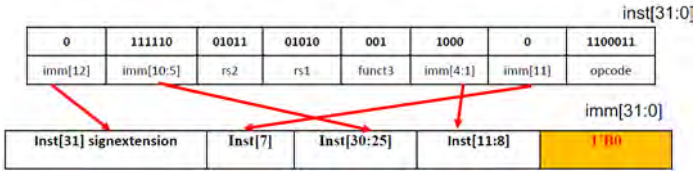


Figure 20: Branch Addressing

$$\text{Target address} = \text{PC} + \text{Branch offset}$$
$$= \text{PC} + \text{immediate} \times 2$$

10.3 Jump Addressing

Jump and link (jal), target uses 20-bit immediate for larger range.

UJ format: 2000=(0 00000000 0 111 1101 0000)

```
jal x0, 2000
```

10.4 Instructions Addressing and their Offset

P85 学会看表

11. Summary

11.1 RISC-V architecture

14 15 16

11.2 RISC-V instruction encoding

17

11.3 RISC-V assembly language

18

12. Synchronization in RISC-V

Two processors sharing an area of memory. Hardware support required. Could be a single instruction. (先读后写)

- Load reserved:

```
lr.d rd,(rs1)
```

Load from address in rs1 to rd. Place reservation on memory address.

- Store conditional:

Table 14: RISC-V operands

Name	Example	Comments
32 registers	x0-x31	Fast locations for data. In RISC-V, data must be in registers to perform arithmetic. Register x0 always equals 0.
2^{61} memory words	Memory[0], Memory[8], ..., Memory[18,446,744,073,709,551,608]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential doubleword accesses differ by 8. Memory holds data structures, arrays, and spilled registers.

Table 15: RISC-V register conventions

Name	Register number	Usage	Preserved on call?
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5-x7	5–7	Temporaries	no
x8-x9	8–9	Saved	yes
x10-x17	10–17	Arguments/results	no
x18-x27	18–27	Saved	yes
x28-x31	28–31	Temporaries	no

Table 16: RISC-V Instruction Format

Name	Field						Comments
(Field size)	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

Table 17: RISC-V instruction encoding

Format	Instruction	Opcode	Funct3	Funct6/7
R-type	add	0110011	000	0000000
	sub	0110011	000	0100000
	sll	0110011	001	0000000
	xor	0110011	100	0000000
	srl	0110011	101	0000000
	sra	0110011	101	0000000
	or	0110011	110	0000000
	and	0110011	111	0000000
	lr.d	0110011	011	0001000
	sc.d	0110011	011	0001100
I-type	lb	0000011	000	n.a.
	lh	0000011	001	n.a.
	lw	0000011	010	n.a.
	ld	0000011	011	n.a.
	lbu	0000011	100	n.a.
	lhu	0000011	101	n.a.
	lwu	0000011	110	n.a.
	addi	0010011	000	n.a.
	slli	0010011	001	000000
	xori	0010011	100	n.a.
	srli	0010011	101	000000
	srai	0010011	101	010000
	ori	0010011	110	n.a.
	andi	0010011	111	n.a.
	jalr	1100111	000	n.a.
S-type	sb	0100011	000	n.a.
	sh	0100011	001	n.a.
	sw	0100011	010	n.a.
	sd	0100011	111	n.a.
SB-type	beq	1100111	000	n.a.
	bne	1100111	001	n.a.
	blt	1100111	100	n.a.
	bge	1100111	101	n.a.
	bltu	1100111	110	n.a.
	bgeu	1100111	111	n.a.
U-type	lui	0110111	n.a.	n.a.
UJ-type	jal	1101111	n.a.	n.a.

Table 18: RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	`add x5, x6, x7`	`x5 = x6 + x7`	Three register operands
	Subtract	`sub x5, x6, x7`	`x5 = x6 - x7`	Three register operands
	Add immediate	`addi x5, x6, 20`	`x5 = x6 + 20`	Used to add constants
	Set if less than	`slt x5, x6, x7`	`x5 = 1 if x5 < x6, else 0`	Three register operands
	Set if less than, unsigned	`sltu x5, x6, x7`	`x5 = 1 if x5 < x6, else 0`	Three register operands
	Set if less than, immediate	`slti x5, x6, x7`	`x5 = 1 if x5 < x6, else 0`	Comparison with immediate
	Set if less than immediate, uns.	`sltiu x5, x6, x7`	`x5 = 1 if x5 < x6, else 0`	Comparison with immediate
	Multiply	`mul x5, x6, x7`	`x5 = x6 × x7`	Lower 64 bits of 128-bit product
	Multiply high	`mulh x5, x6, x7`	`x5 = (x6 × x7) >> 64`	Upper 64 bits of 128-bit signed product
	Multiply high, unsigned	`mulhu x5, x6, x7`	`x5 = (x6 × x7) >> 64`	Upper 64 bits of 128-bit unsigned product
	Multiply high, signed-unsigned	`mulhsu x5, x6, x7`	`x5 = (x6 × x7) >> 64`	Upper 64 bits of 128-bit signed-unsigned product
	Divide	`div x5, x6, x7`	`x5 = x6 / x7`	Divide signed 64-bit numbers
	Divide unsigned	`divu x5, x6, x7`	`x5 = x6 / x7`	Divide unsigned 64-bit numbers
	Remainder	`rem x5, x6, x7`	`x5 = x6 % x7`	Remainder of signed 64-bit division
	Remainder unsigned	`remu x5, x6, x7`	`x5 = x6 % x7`	Remainder of unsigned 64-bit division
Data transfer	Load doubleword	`ld x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Doubleword from memory to register
	Store doubleword	`sd x5, 40(x6)`	`Memory[x6 + 40] = x5`	Doubleword from register to memory
	Load word	`lw x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Word from memory to register
	Load word, unsigned	`lwu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned word from memory to register
	Store word	`sw x5, 40(x6)`	`Memory[x6 + 40] = x5`	Word from register to memory
	Load halfword	`lh x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Halfword from memory to register
	Load halfword, unsigned	`lhu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned halfword from memory to register
	Store halfword	`sh x5, 40(x6)`	`Memory[x6 + 40] = x5`	Halfword from register to memory
	Load byte	`lb x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte from memory to register
	Load byte, unsigned	`lbu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte halfword from memory to register
	Store byte	`sb x5, 40(x6)`	`Memory[x6 + 40] = x5`	Byte from register to memory
	Load reserved	`lr.d x5, (x6)`	`x5 = Memory[x6]`	Load; 1st half of atomic swap
	Store conditional	`sc.d x7, x5, (x6)`	`Memory[x6] = x5; x7 = 0/1`	Store; 2nd half of atomic swap
	Load upper immediate	`lui x5, 0x12345`	`x5 = 0x12345000`	Loads 20-bit constant shifted left 12 bits
	Add upper immediate to PC	`auipc x5, 0x12345`	`x5 = PC + 0x12345000`	Used for PC-relative data addressing
Logical	And	`and x5, x6, x7`	`x5 = x6 & x7`	Three reg. operands; bit-by-bit AND
	Inclusive or	`or x5, x6, x8`	`x5 = x6 | x8`	Three reg. operands; bit-by-bit OR
	Exclusive or	`xor x5, x6, x9`	`x5 = x6 ^ x9`	Three reg. operands; bit-by-bit XOR
	And immediate	`andi x5, x6, 20`	`x5 = x6 & 20`	Bit-by-bit AND reg. with constant
	Inclusive or immediate	`ori x5, x6, 20`	`x5 = x6 | 20`	Bit-by-bit OR reg. with constant
	Exclusive or immediate	`xori x5, x6, 20`	`x5 = x6 ^ 20`	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	`sll x5, x6, x7`	`x5 = x6 << x7`	Shift left by register
	Shift right logical	`srl x5, x6, x7`	`x5 = x6 >> x7`	Shift right by register
	Shift right arithmetic	`sra x5, x6, x7`	`x5 = x6 >> x7`	Arithmetic shift right by register
	Shift left logical immediate	`slli x5, x6, 3`	`x5 = x6 << 3`	Shift left by immediate
	Shift right logical immediate	`srli x5, x6, 3`	`x5 = x6 >> 3`	Shift right by immediate
	Shift right arithmetic immediate	`srai x5, x6, 3`	`x5 = x6 >> 3`	Arithmetic shift right by immediate
Conditional branch	Branch if equal	`beq x5, x6, 100`	`if (x5 == x6) go to PC+100`	PC-relative branch if registers equal
	Branch if not equal	`bne x5, x6, 100`	`if (x5 != x6) go to PC+100`	PC-relative branch if registers not equal
	Branch if less than	`blt x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less
	Branch if greater or equal	`bge x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal
	Branch if less, unsigned	`bltu x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less
	Branch if greatr/eq, unsigned	`bgeu x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal
Uncondit-ional branch	Jump and link	`jal x1, 100`	`x1 = PC+4; go to PC+100`	PC-relative procedure call
	Jump and link register	`jalr x1, 100(x5)`	`x1 = PC+4; go to x5+100`	Procedure return; indirect call

```
sc.d rd,(rs1),rs2
```

Store from rs2 to address in rs1. Succeeds if location not changed since the lr.d. Returns 0 in rd. Fails if location is changed. Returns non-zero value in rd

12.1 atomic swap (to test/set lock variable)

RISC-V assembly code

```
again:
    lr.d x10, (x20)
    sc.d x11, (x20), x23
    bne x11, x0, again
    addi x23, x10, 0
```

1) x11 = status
2) branch if store failed
3) x23 = loaded value

12.2 lock

RISC-V assembly code

```
    addi x12, x0, 1
again:
    lr.d x10, (x20)
    bne x10, x0, again
    sc.d x11, (x20), x12
    bne x11, x0, again
```

1) copy locked value
2) read lock
3) check if it is 0 yet
4) attempt to store
5) branch if fails

RISC-V assembly code

```
Unlock:
    sd x0, 0(x20)
```

free lock

13. Translating and starting a program

13.1 Producing an Object Module

Assembler (or compiler) translates program into machine instructions. Provides information for building a complete program from the pieces:

- Header: described contents of object module

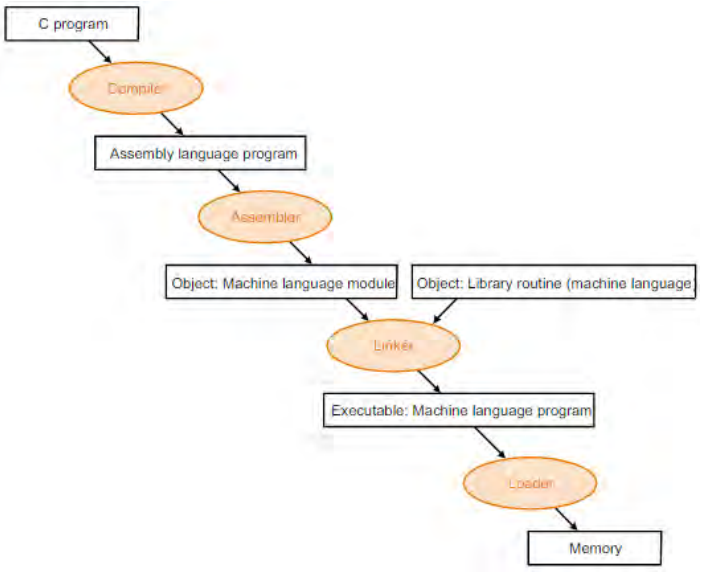


Figure 21: Translating a program

- Text segment: translated instructions
- Static data segment: data allocated for the life of the program
- Relocation info: for contents that depend on absolute location of loaded program
- Symbol table: global definitions and external refs
- Debug info: for associating with source code

Table 19: Producing an Object Module

Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	ld x10, 0(x3)	
	4	jal x1, 0	
	...	...	
Data segment	0	(X)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	ld	X
	4	jal	B
Symbol table	Label	Address	
	X	–	
	B	–	

13.2 Link

Object modules(including library routine) ← executable program.

1) Place code and data modules symbolically in memory

2) Determine the addresses of data and instruction labels

3) Patch both the internal and external references (Address of invoke)

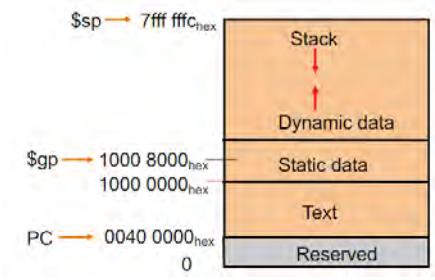


Figure 22: Link

13.3 Loading a Program

Load from image file (镜像文件) on disk into memory

1) Read header to determine segment sizes
2) Create virtual address space
3) Copy text and initialized data into memory. Or set page table entries so they can be faulted in
4) Set up arguments on stack
5) Initialize registers (including sp, fp, gp)
6) Jump to startup routine

- Copies arguments to x10, ... and calls main
- When main returns, do exit syscall

13.4 Dynamic Linking

Only link/load library procedure when it is called.

- Requires procedure code to be relocatable
- Avoids image bloat caused by static linking of all (transitively) referenced libraries
- Automatically picks up new library versions

13.5 Lazy Linkage

14. A C Sort Example To Put it All Together

Three general steps for translating C procedures

1) Allocate registers to program variables
2) Produce code for the body of the procedures
3) Preserve registers across the procedures invocation

P104

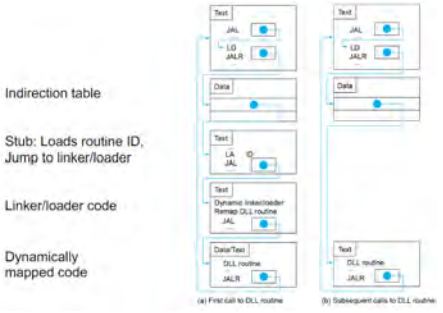


Figure 23: Lazy Linkage

15. Arrays versus Pointers

e.g. P111

15.1 Comparison of Array vs Pointers

P112

IV The Processor

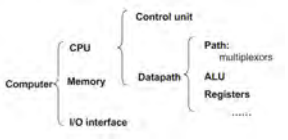


Figure 24: Comptuer Organization

1. Introduction

- CPU performance factors
  - Instruction count: Determined by ISA and compiler
  - CPI and Cycle time: Determined by CPU hardware
- implementations
  - A simplified version
  - A more realistic pipelined version
- Simple subset, shows most aspects

1.1 An overview of Implementation

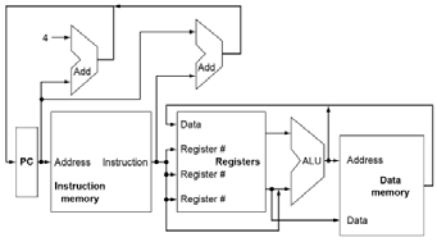


Figure 25: An overview of Implementation

Must use multiplexers to join wires together.

1.2 Control

Control the units: Read Memory, Write Memory. (blue wires are control)

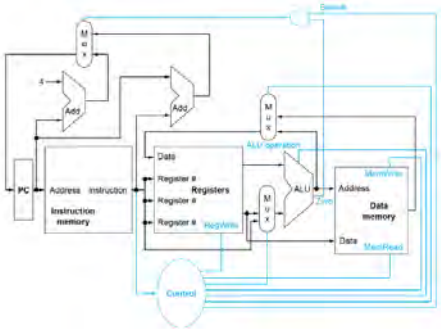


Figure 26: Control

2. Logic Design Conventions

- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements: Store information

2.1 Clocking Methodology

Combinational logic transforms data during clock cycles.

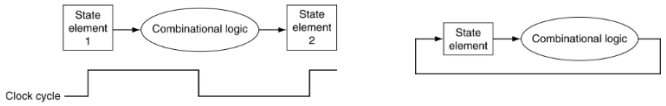


Figure 27: Clocking Methodology

3. Building a Datapath

Datapath is elements that process data and addresses in the CPU.

3.1 Instruction execution in RISC-V

1) Fetch :
   a. Take instructions from the instruction memory
   b. Modify PC to point the next instruction
2) Instruction decoding & Read Operand:
   a. Will be translated into machine control command
   b. Reading Register Operands, whether or not to use
   c. Reading Register Operands, whether or not to use
3) Executive Control:
   a. Control the implementation of the corresponding ALU operation
4) Memory access:
   a. Write or Read data from memory
   b. Only ld/sd
5) Write results to register:
   a. If it is R-type instructions, ALU results are written to rd
   b. If it is I-type instructions, memory data are written to rd

- Modify PC for branch instructions: Can be everywhere

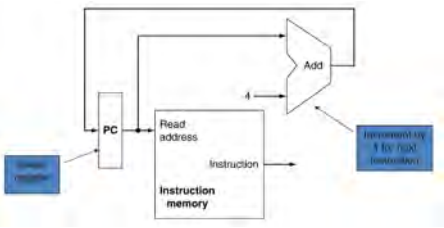


Figure 28: Instruction Fetch

3.2 Instruction Fetch

[28]

3.3 R-Format Instructions

1) Read two register operands
2) Perform arithmetic/logical operation
3) Write register result

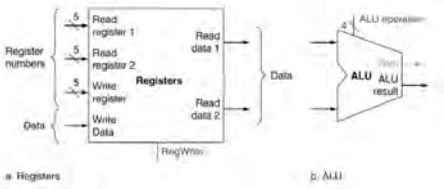


Figure 29: R-Format Instructions

3.4 Load/Store Instructions

1) Read register operands
2) Calculate address using 12-bit offset

Use ALU, but sign-extend offset

3) Load: Read memory and update register
4) Store: Write register value to memory

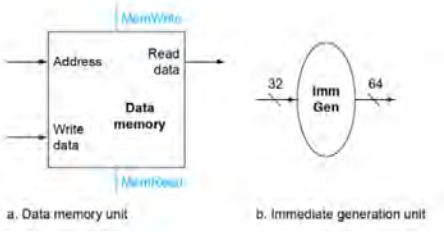


Figure 30: Load/Store Instructions

3.5 Branch Instructions

1) Read register operands
2) Compare operands

Use ALU, subtract and check Zero output

3) Calculate target address
   a. Sign-extend displacement
   b. Shift left 1 place (halfword displacement)
   c. Add to PC value

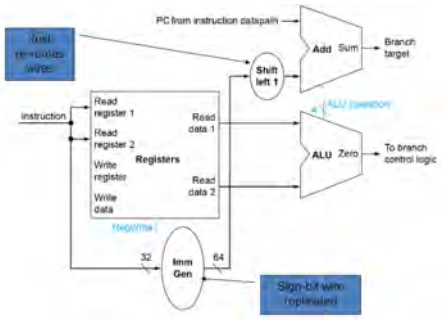


Figure 31: Branch Instructions

3.6 Composing the Elements

1) First-cut data path does an instruction in one clock cycle. Each datapath element can only do one function at a time. Hence, we need separate instruction and data memories.

2) Use multiplexers where alternate data sources are used for different instructions

3.7 Path Built using Multiplexer

- R-type instruction Datapath
- I-type instruction Datapath
  - For ALU
  - For load
- S-type (store) instruction Datapath
- SB-type (branch) instruction Datapath
- UJ-type instruction Datapath
  - For Jump

1) R type Instruction & Data stream
2) I type Instruction & Data stream

(下方 Sign extend 可以直接为 imm Gen)

3) S type Instruction & Data stream
4) SB type Instruction & Data stream

(最右上角 PC from instruction datapath 不过 add 的需要 PC+4)

5) Jal/J type Instruction & Data stream

(这个图不太对)

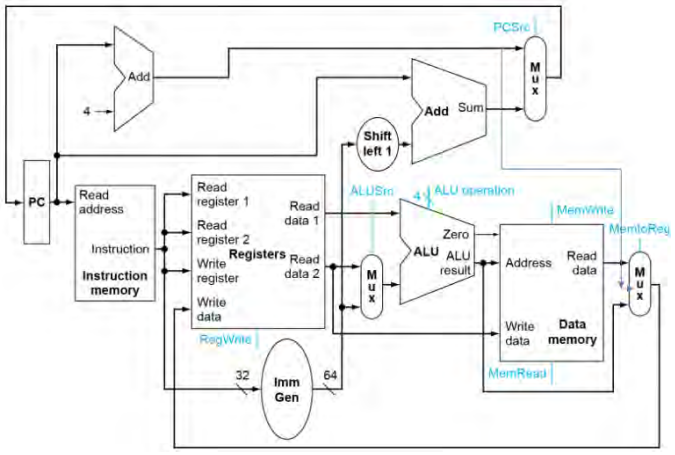


Figure 32: Full Datapath

3.8 Full Datapath

32

4. A Simple Implementation Scheme

4.1 Building Controller

There are 7+4 signals

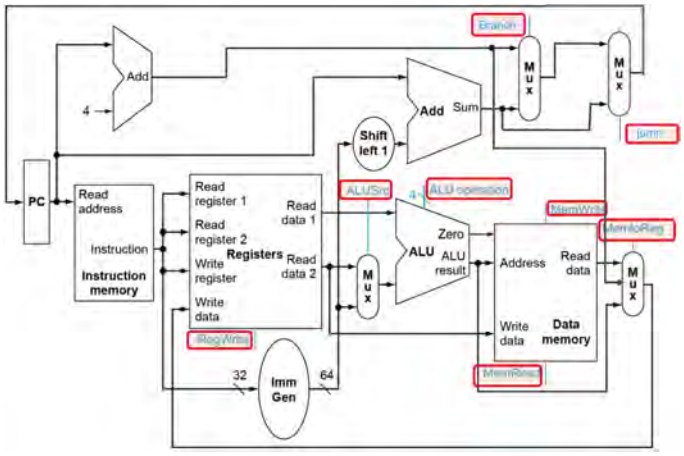


Figure 33: Building Controller

4.2 Scheme of Controller

34

4.3 Designing the Main Control Unit — First level

Main Control Unit function

1) ALU op (2)
2) Divided 7 control signals into 2 groups
   - 4 Mux

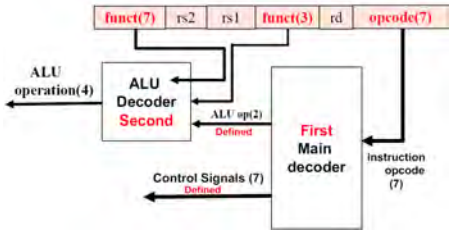


Figure 34: Scheme of Controller

- 3 R/W

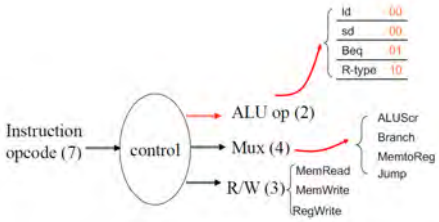


Figure 35: Main Control Unit

Table 20: Truth tables of main Controller

輸入		輸出								
Instruction	OPCode	ALUSrcB	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	Jump	ALU Op1	ALU Op0
R-format	0110011	0	00	1	0	0	0	0	1	0
Ld(I-Type)	0000011	1	01	1	1	0	0	0	0	0
sd(S-Type)	0100011	1	X	0	0	1	0	0	0	0
beq(SB-Type)	1100111	0	X	0	0	0	1	0	0	1
Jal(UJ-Type)	1101111	X	10	1	0	0	0	1	X	X

4.4 Design the ALU Decoder second level

ALU used for

1) Load/Store: F = add
2) Branch: F = subtract
3) R-type: F depends on opcode

Table 21: ALU Decoder

opcode	ALUOp	Operation	Funct7	funct3	ALU function	ALU control
ld	00	load register	XXXXXXX	xxx	add	0010
sd	00	store register	XXXXXXX	xxx	add	0010
beq	01	branch on equal	XXXXXXX	xxx	subtract	0110
R-type	10	add	0000000	000	add	0010
		subtract	0100000	000	subtract	0110
		AND	0000000	111	AND	0000
		OR	0000000	110	OR	0001
		SLT	0000000	010	Slt	0111

4.5 Datapath with Control

36

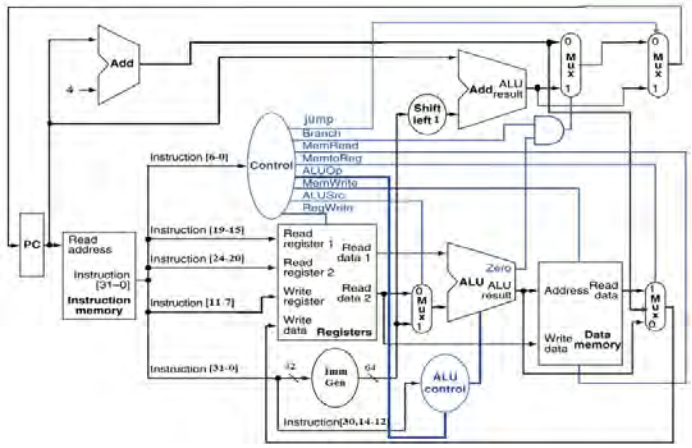


Figure 36: Datapath with Control

5. An overview of pipelining

5.1 Performance Issues

Longest delay of Critical path (load instruction) determines clock period. Wihout violating design principle: Making the common case fast, we will improve performance by pipelining

5.1.1 Pipelining Analogy Pipelined laundry: overlapping execution

5.2 RISC-V Pipeline

Five stages:

1) IF: Instruction fetch from memory
2) ID: Instruction decode & register read
3) EX: Execute operation or calculate address
4) MEM: Access memory operand
5) WB: Write result back to register

5.2.1 Pipelining RISC-V instruction set CPI is decreased to 1, since one instruction will be issued (or finished) each cycle.

During any cycle, one instruction is present in each stage.

	Clock Number								
	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EX	MEM	WB				
Instruction i+1		IF	ID	EX	MEM	WB			
Instruction i+2			IF	ID	EX	MEM	WB		
Instruction i+3				IF	ID	EX	MEM	WB	
Instruction i+4					IF	ID	EX	MEM	WB

Figure 37: Pipelining RISC-V instruction set

Ideally, performance is increased five fold!

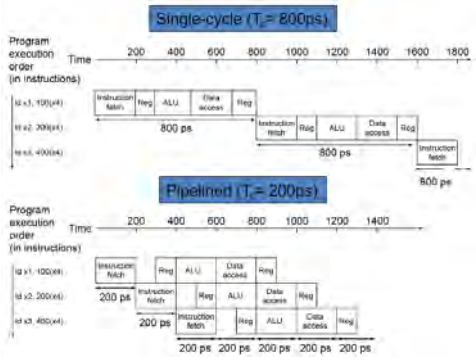


Figure 38: Pipeline Performance

5.2.2 Pipeline Performance Reg 滞后是防止读写冲突.

5.2.3 Pipeline Speedup If all stages are balanced, i.e., all take the same time.

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instruction}_{\text{pipelined}}}{\text{Number of stages}}$$

If not balanced, speedup is less.

Speedup due to increased throughput n Latency (time for each instruction) does not decrease

5.2.4 Pipelining and ISA Design RISC-V ISA designed for pipelining

1) All instructions are 32-bits
   Easier to fetch and decode in one cycle
   c.f. x86: 1- to 17-byte instructions
2) Few and regular instruction formats
   Can decode and read registers in one step
3) Load/store addressing
   Can calculate address in 3rd stage, access memory in 4th stage

5.3 Hazards

Situations that prevent starting the next instruction in the next cycle.

1) **Structure hazards**: A required resource is busy
2) **Data hazard**: Need to wait for previous instruction to complete its data read/write
3) **Control hazard**: Deciding on control action depends on previous instruction

5.4 Structure Hazards

Conflict for use of a resource.

In RISC-V pipeline with a single memory, Load/store requires data access but instruction fetch would have to stall for that cycle which would cause a pipeline "bubble".

Hence, pipelined datapaths require separate instruction/data memories, or separate instruction/data caches.

5.5 Data Hazard

An instruction depends on completion of data access by a previous instruction.

```
1 add x19, x0, x1
2 sub x2, x19, x3
```

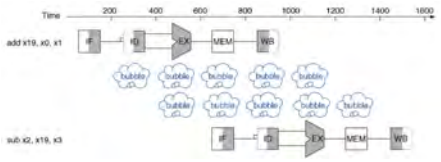


Figure 39: Data Hazard

5.5.1 Forwarding (aka Bypassing) Use result when it is computed. Don't wait for it to be stored in a register. But requires extra connections in the datapath.

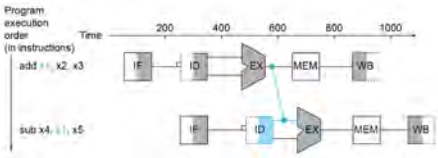


Figure 40: Forwarding

5.5.2 Load-Use Data Hazard Can't always avoid stalls by forwarding. If value not computed when needed, can't forward backward in time!

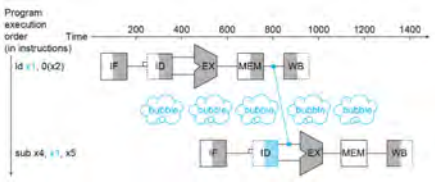


Figure 41: Load-Use Data Hazard

5.5.3 Code Scheduling to Avoid Stalls Reorder code to avoid use of load result in the next instruction.

```
1 a=b+e;
2 c=b+f;
```

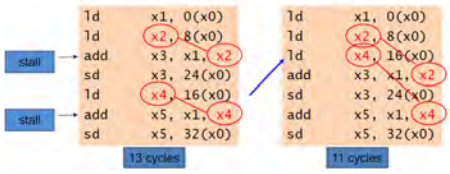


Figure 42: Code Scheduling to Avoid Stalls

5.6 Control Hazards

Branch determines flow of control. Fetching next instruction depends on branch outcome. Pipeline can't always fetch correct instruction, Still working on ID stage of branch.

In RISC-V pipeline, need to compare registers and compute target early in the pipeline, add hardware to do it in ID stage.

5.6.1 Stall on Branch Wait until branch outcome determined before fetching next instruction.

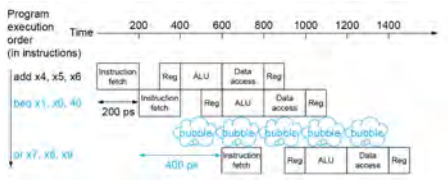


Figure 43: Stall on Branch

5.6.2 Branch Prediction Longer pipelines can't readily determine branch outcome early. Can predict outcome of branch, only stall if prediction is wrong.

In RISC-V pipeline, Can predict branches not taken and Fetch instruction after branch, with no delay

5.6.3 More-Realistic Branch Prediction

- **Static branch prediction**: Based on typical branch behavior
- **Dynamic branch prediction**: Hardware measures actual branch behavior, assume future behavior will continue the trend.

5.7 Pipeline Summary

Pipelining improves performance by increasing instruction throughput.

- Executes multiple instructions in parallel
- Each instruction has the same latency

Subject to hazards: Structure, data, control

Instruction set design affects complexity of pipeline implementation.

6. RISC-V Pipelined Datapath

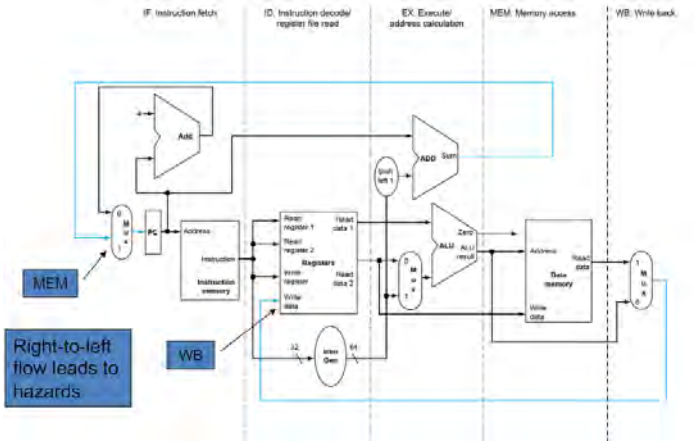


Figure 44: RISC-V Pipelined Datapath

6.1 Pipeline registers

To hold information produced in previous cycle, need registers between stages.

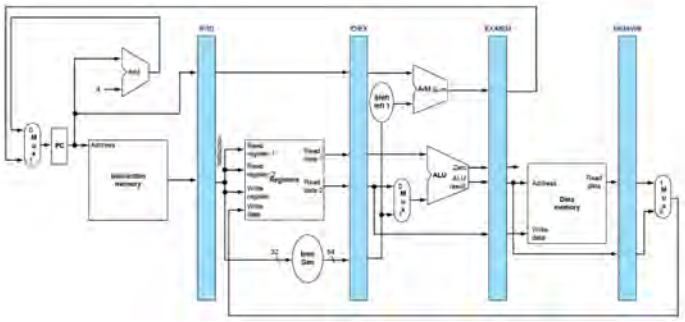


Figure 45: Pipeline registers

6.2 Pipeline Operation

Cycle-by-cycle flow of instructions through the pipelined datapath.

- "Single-clock-cycle" pipeline diagram: Shows pipeline usage in a single cycle and Highlight resources used
- c.f. "multi-clock-cycle" diagram: Graph of operation over time

For Load

1) IF
2) ID
3) EX
4) MEM
5) WB
6) Corrected Datapath

6.2.1 Multi-Cycle Pipeline Diagram Form showing resource usage

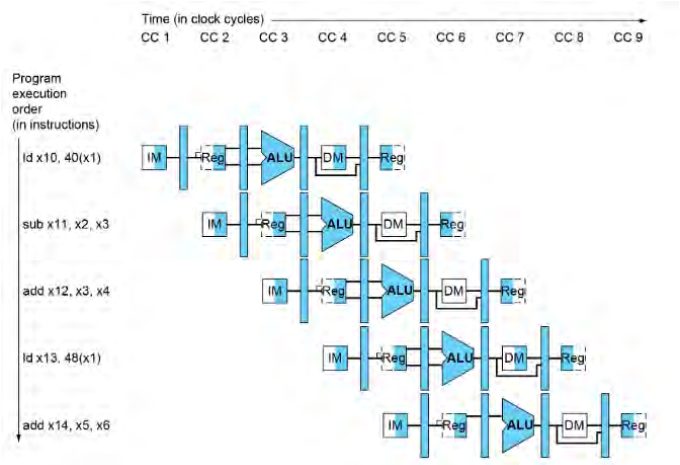


Figure 46: Multi-Cycle Pipeline Diagram

6.2.2 Single-Cycle Pipeline Diagram State of pipeline in a given cycle

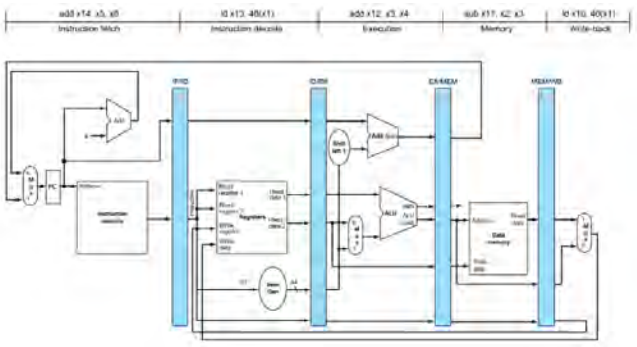


Figure 47: Single-Cycle Pipeline Diagram

6.3 Pipelined Control

As in single-cycle implementation, control signals derived from instruction.

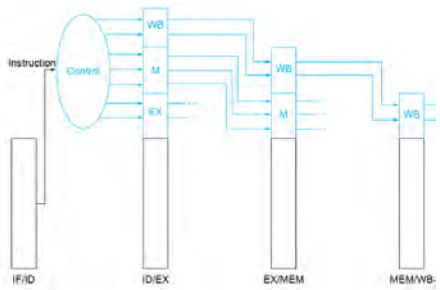


Figure 48: control signals

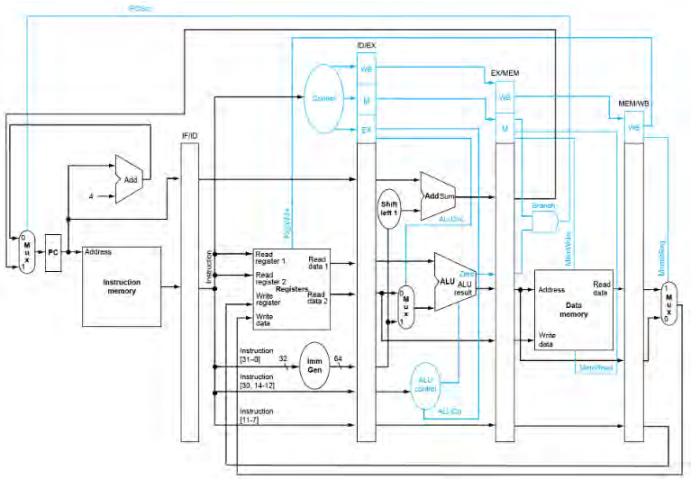


Figure 49: Pipelined Control

7. Data Hazards

Consider this sequence:

```
1 sub x2, x1,x3
2 and x12,x2,x5
3 or x13,x6,x2
4 add x14,x2,x2
5 sd x15,100(x2)
```

7.1 Dependencies and Forwarding

7.1.1 Detecting the Need to Forward Pass register numbers along pipeline.

e.g., ID/EX.RegisterRs1 = register number for Rs1 sitting in ID/EX pipeline register.

ALU operand register numbers in EX stage are given by ID/EX.RegisterRs1, ID/EX.RegisterRs2.

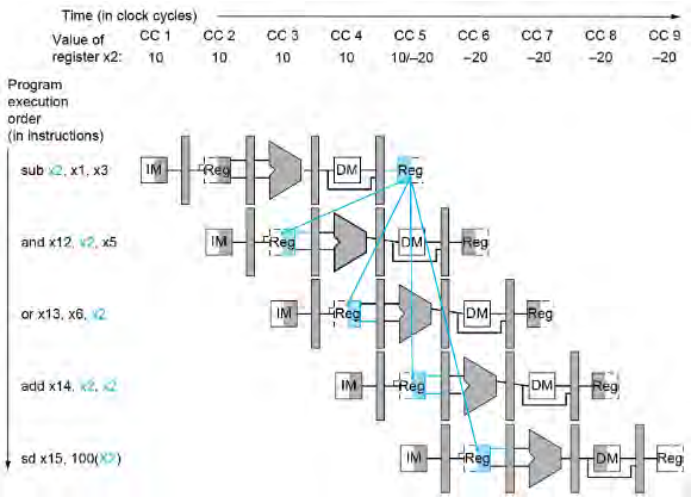


Figure 50: Dependencies and Forwarding

Data hazards when

$$\left.\begin{array}{l}\text{EX/MEM.RegisterRd} = \text{ID/EX.RegisterRs1}\\ \text{EX/MEM.RegisterRd} = \text{ID/EX.RegisterRs2}\end{array}\right\}\ \text{Fwd from EX/MEM pipeline reg}$$

$$\left.\begin{array}{l}\text{MEM/WB.RegisterRd} = \text{ID/EX.RegisterRs1}\\ \text{MEM/WB.RegisterRd} = \text{ID/EX.RegisterRs2}\end{array}\right\}\ \text{Fwd from MEM/WB pipeline reg}$$

But only if forwarding instruction will write to a register! EX/MEM.RegWrite, MEM/WB.RegWrite.

And only if Rd for that instruction is not x0.

$$\text{EX/MEM.RegisterRd} \neq 0$$
$$\text{MEM/WB.RegisterRd} \neq 0$$

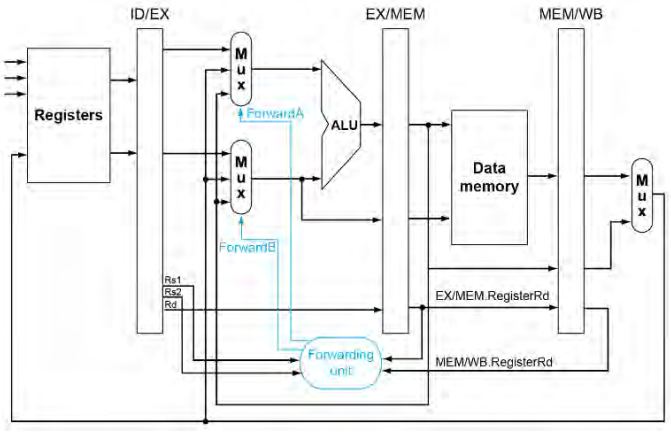


Figure 51: Forwarding Paths

Table 22: Forwarding Conditions

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

7.2 Double Data Hazard

Consider the sequence:

```
add x1,x1,x2
add x1,x1,x3
add x1,x1,x4
```

Revise MEM hazard condition, only fwd if EX hazard condition isn't true.

7.2.1 Revised Forwarding Condition MEM hazard

```
if (
    MEM/WB.RegWrite
    and (MEM/WB.RegisterRd != 0)
    and not(
        EX/MEM.RegWrite
        and (EX/MEM.RegisterRd != 0)
        and (EX/MEM.RegisterRd = ID/EX.
            RegisterRs1)
    )
    and (MEM/WB.RegisterRd = ID/EX.
        RegisterRs1)
) ForwardA = 01;
```

RegisterRs1 is same.

7.3 Load-Use Hazard Detection

Check when using instruction is decoded in ID stage.

ALU operand register numbers in ID stage are given by IF/ID.RegisterRs1 or IF/ID.RegisterRs2.

Load-use hazard when

```
ID/EX.MemRead and (
    (ID/EX.RegisterRd = IF/ID.RegisterRs1) or
    (ID/EX.RegisterRd = IF/ID.RegisterRs2)
)
```

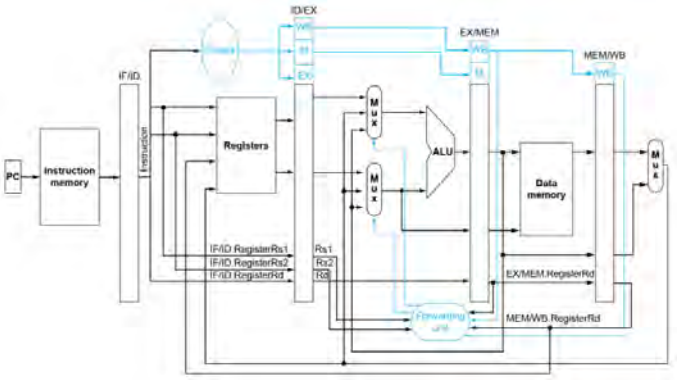


Figure 52: Datapath with Forwarding

If detected, stall and insert bubble.

7.3.1 How to Stall the Pipeline Force control values in ID/EX register to 0, i.e. EX, MEM and WB do nop (no-operation).

Prevent update of PC and IF/ID register.

1) Using instruction is decoded again
2) Following instruction is fetched again
3) 1-cycle stall allows MEM to read data for ld
   Can subsequently forward to EX stage

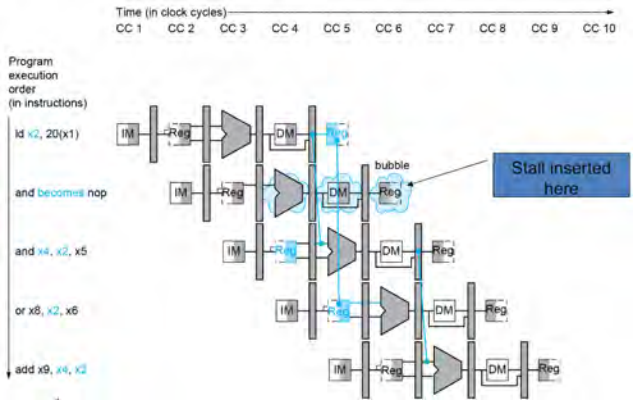


Figure 53: Load-Use Data Hazard

7.4 Stalls and Performance

Stalls reduce performance, but are required to get correct results.

Compiler can arrange code to avoid hazards and stalls.And requires knowledge of the pipeline structure.

8. Branch Hazards

If branch outcome determined in MEM, we will lose three instructions

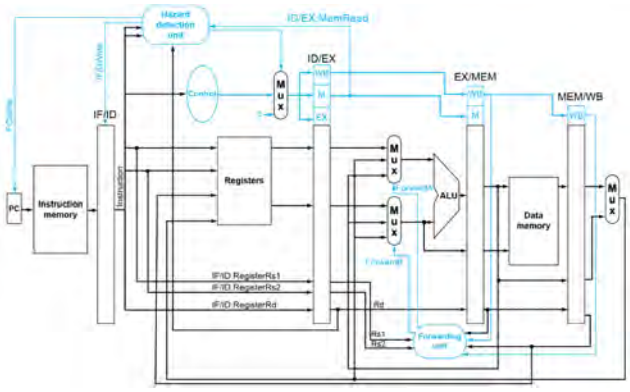


Figure 54: Datapath with Hazard Detection

8.1 Reducing Branch Delay

Move hardware to determine outcome to ID stage.
Example: branch taken

```
36: sub x10, x4, x8
40: beq x1, x3, 16
44: and x12, x2, x5
48: or  x13, x2, x6
52: add x14, x4, x2
56: sub x15, x6, x7
...
72: ld x4, 50(x7)
```

PC-relative branch to 40+16*2=72

8.2 Dynamic Branch Prediction

Branch prediction buffer (aka branch history table) indexed by recent branch instruction addresses stores outcome (taken/not taken). To execute a branch

1) Check table, expect the same outcome
2) Start fetching from fall-through or target
3) If wrong, flush pipeline and flip prediction

8.2.1 1-Bit Predictor: Shortcoming Inner loop branches mispredicted twice.

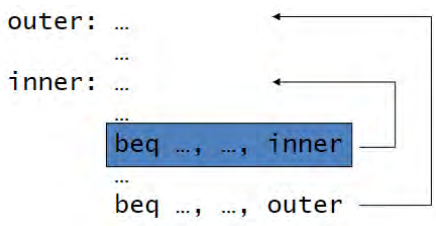


Figure 55: Inner loop branches

Mispredict as taken on last iteration of inner loop. Then mispredict as not taken on first iteration of inner loop next time around.

8.2.2 2-Bit Predictor Only change prediction on two successive mispredictions.

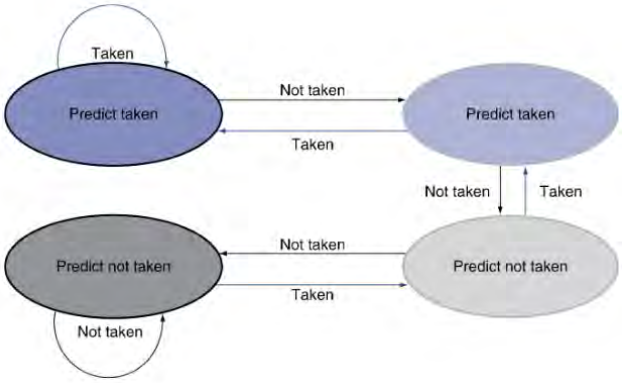


Figure 56: 2-Bit Predictor

8.2.3 Calculating the Branch Target Even with predictor, still need to calculate the target address, 1-cycle penalty for a taken branch.

Branch target buffer cache of target addresses Indexed by PC when instruction fetched. If hit and instruction is branch predicted taken, can fetch target immediately.

9. Exceptions and Interrupts

Exception: Arises within the CPU
Interrupt: From an external I/O controller

9.1 Handling Exceptions

1) Save PC of offending (or interrupted) instruction. In RISC-V: Supervisor Exception Program Counter (SEPC)
2) Save indication of the problem. In RISC-V: Supervisor Exception Cause Register (SCAUSE)
3) Jump to handler

9.1.1 An Alternate Mechanism Vectored Interrupts:
Handler address determined by the cause.

Exception vector address to be added to a vector table base register:

- Undefined opcode: 00 0100 0000_2
- Hardware malfunction: 01 1000 0000_2
- ...

Instructions either Deal with the interrupt, or Jump to real handler.

9.2 Handler Actions

1) Read cause, and transfer to relevant handler.
2) Determine action required
3) If restartable
   Take corrective action
   use SEPC to return to program
4) Otherwise
   Terminate program
   Report error using SEPC, SCAUSE, ...

9.3 Exceptions in a Pipeline

Exceptions is Another form of control hazard.

Similar to mispredicted branch, Use much of the same hardware.

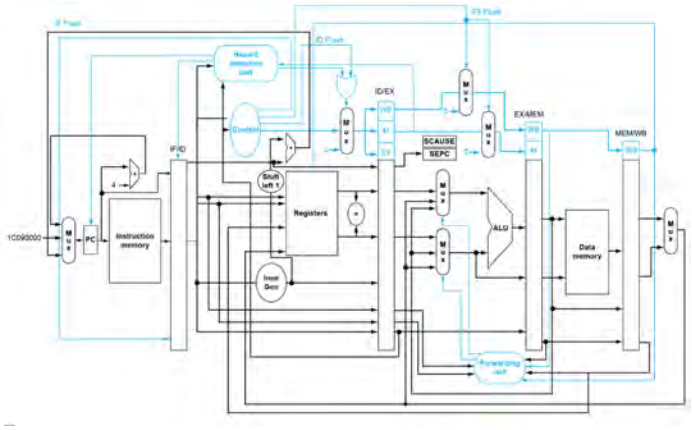


Figure 57: Pipeline with Exceptions

9.3.1 Exception Properties Restartable exceptions:

- Pipeline can flush the instruction
- Handler executes, then returns to the instruction
  Refetched and executed from scratch

PC saved in SEPC register Identifies causing instruction.

9.3.2 Multiple Exceptions Pipelining overlaps multiple instructions could have multiple exceptions at once.

Simple approach ("Precise" exceptions): deal with exception from earliest instruction, Flush subsequent instructions.

In complex pipelines, Multiple instructions issued per cycle, Out-of-order completion, Maintaining precise exceptions is difficult!

9.3.3 Imprecise Exceptions Just stop pipeline and save state including exception cause(s).

Let the handler work out which instruction(s) had exceptions which to complete or flush. May require "manual" completion.

Simplifies hardware, but more complex handler software.

Not feasible for complex multiple-issue out-of-order pipelines.

10. Instruction-Level Parallelism (ILP)

Pipelining: executing multiple instructions in parallel.

To increase ILP:

1) Deeper pipeline
   Less work per stage ⇒ shorter clock cycle.
2) Multiple issue
   Replicate pipeline stages ⇒ multiple pipelines
   Start multiple instructions per clock cycle
   CPI < 1, so use Instructions Per Cycle (IPC)

But dependencies reduce this in practice

10.1 Multiple Issue

- Static multiple issue: Compiler groups instructions to be issued together, then packages them into "issue slots". It's compiler that detects and avoids hazards.
- Dynamic multiple issue: CPU examines instruction stream and chooses instructions to issue each cycle. Compiler can help by reordering instructions. CPU resolves hazards using advanced techniques at runtime.

10.2 Speculation

"Guess" what to do with an instruction

1) Start operation as soon as possible
2) Check whether guess was right
   If so, complete the operation
   If not, roll-back and do the right thing

Common to static and dynamic multiple issue.

e.g.

- Speculate on branch outcome
  Roll back if path taken is different.
- Speculate on load
  Roll back if location is updated.

10.2.1 Compiler Speculation Compiler can reorder instructions, which can include "fix-up" instructions to recover from incorrect guess. e.g., move load before branch.

10.2.2 Hardware Speculation Hardware can look ahead for instructions to execute. Buffer results until it determines they are actually needed. Flush buffers on incorrect speculation.

10.2.3 Speculation and Exceptions If exception occurs on a speculatively executed instruction,

- Static speculation: Can add ISA support for deferring exceptions.
- Dynamic speculation: Can buffer exceptions until instruction completion (which may not occur).

10.3 Static Multiple Issue

Compiler groups instructions into "issue packets", which is group of instructions that can be issued on a single cycle and determined by pipeline resources required.

Think of an issue packet as a very long instruction (Very Long Instruction Word (VLIW)), which Specifies multiple concurrent operations .

10.3.1 Scheduling Static Multiple Issue Compiler must remove some/all hazards

1) Reorder instructions into issue packets
2) No dependencies with a packet
3) Possibly some dependencies between packets
   Varies between ISAs; compiler must know!
4) Pad with nop if necessary

10.3.2 RISC-V with Static Dual Issue Two-issue packets

- One ALU/branch instruction
- One load/store instruction
- 64-bit aligned
  ALU/branch, then load/store
  Pad an unused instruction with nop

10.3.3 Hazards in the Dual-Issue RISC-V More instructions executing in parallel. And more aggressive scheduling required

- EX data hazard
  Forwarding avoided stalls with single-issue
  Now can't use ALU result in load/store in same packet, should split into two packets, effectively a stall
- Load-use hazard
  Still one cycle use latency, but now two instructions

10.3.4 Scheduling Example Schedule this for dual-issue RISC-V

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

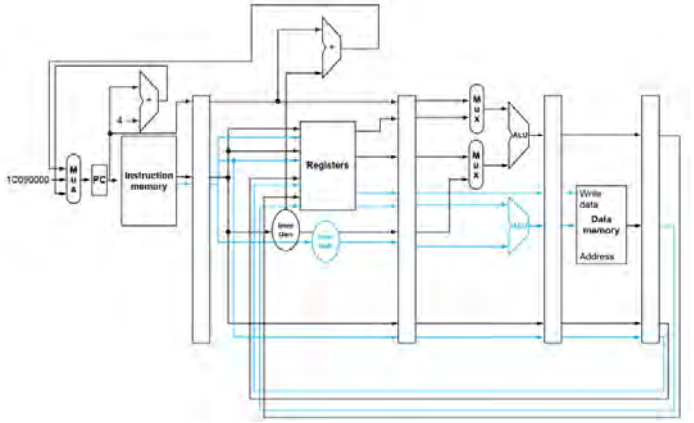


Figure 58: RISC-V with Static Dual Issue

```
1 Loop:
2     ld x31, 0(x20)
3     add x31, x31, x21
4     sd x31, 0(x20)
5     addi x20, x20, -8
6     blt ,22, x20, Loop
```

1) x31=array element
2) add scalar in x21
3) store result
4) decrement pointer
5) branch if x22<x20

	ALU/branch	Load/store	cycle
Loop:	nop	ld x31,0(x20)	1
	addi x20,x20,-8	nop	2
	add x31,x31,x21	nop	3
	blt x22,x20,Loop	sd x31,8(x20)	4

Figure 59: Scheduling Example

IPC = 5/4 = 1.25 (c.f. peak IPC = 2)

10.3.5 Loop Unrolling

- Replicate loop body to expose more parallelism which reduces loop-control overhead.
- Use different registers per replication called "register renaming"

avoid loop-carried "anti-dependencies".

store followed by a load of the same register

Aka "name dependence", reuse of a register name

	ALU/branch	Load/store	cycle
Loop:	addi x20,x20,-32	ld x28, 0(x20)	1
	nop	ld x29, 24(x20)	2
	add x28,x28,x21	ld x30, 16(x20)	3
	add x29,x29,x21	ld x31, 8(x20)	4
	add x30,x30,x21	sd x28, 32(x20)	5
	add x31,x31,x21	sd x29, 24(x20)	6
	nop	sd x30, 16(x20)	7
	blt x22,x20,Loop	sd x31, 8(x20)	8

Figure 60: Loop Unrolling Example

10.3.6 Loop Unrolling Example IPC = 14/8 = 1.75. Closer to 2, but at cost of registers and code size

10.4 Dynamic Multiple Issue

CPU decides whether to issue $0, 1, 2, \ldots$ each cycle, Avoiding structural and data hazards.

Avoids the need for compiler scheduling. Though it may still help, code semantics ensured by the CPU

10.4.1 Dynamic Pipeline Scheduling Allow the CPU to execute instructions out of order to avoid stalls, but commit result to registers in order.

e.g.

```
1    ld x31, 20(x21)
2    add x1, x31, x2
3    sub x23, x23, x3
4    andi x5, x23, 20
```

Can start sub while add is waiting for ld

10.4.2 Dynamically Scheduled CPU [61]

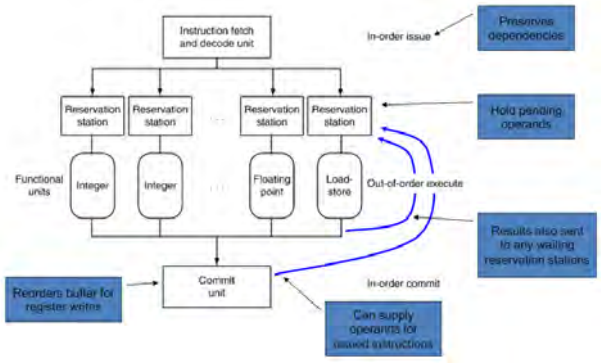


Figure 61: Dynamically Scheduled CPU

10.4.3 Register Renaming Reservation stations and reorder buffer effectively provide register renaming. On instruction issue to reservation station

- If operand is available in register file or reorder buffer
  Copied to reservation station
  No longer required in the register; can be overwritten
- If operand is not yet available
  It will be provided to the reservation station by a function unit
  Register update may not be required

10.4.4 Speculation Predict branch and continue issuing. Don't commit until branch outcome determined.

Load speculation: Avoid load and cache miss delay

- Predict the effective address
- Predict loaded value
- Load before completing outstanding stores
- Bypass stored values to load unit

Don't commit load until speculation cleared

V Large and Fast: Exploiting Memory Hierarchy

1. Large and Fast

- Size large enough to store any program (Turing Completeness)
- Speed fast enough to catch up with the speed of the processor (IF, MEM....)

Reality: we don't have an ideal memory, so we need a hierarchical design.

1.1 Key merits of a memory

1) Speed
   Physical cell property
   Access schemes
2) Cost per bit (cost vs size)
3) Volatility
4) Endurance cycles
   Controller design

1.2 Memory Technologies

Memories: Review

1) SRAM (Static Random Access Memory)
   - value is stored on a pair of inverting gates
   - very fast but takes up more space than DRAM

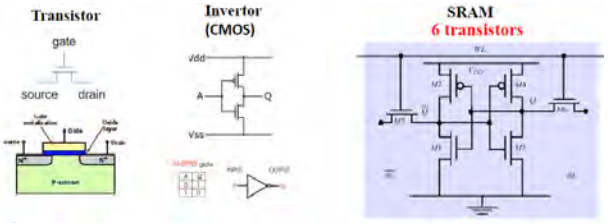


Figure 62: SRAM

2) DRAM (Dynamic Random Access Memory)
   - Value is stored as a charge on capacitor
   - Very small but slower than SRAM (factor of 5 to 10)
   - Must periodically be refreshed
   - Read contents and write back (destructive read)

   Advanced DRAM Organization:

   - Bits in a DRAM are organized as a rectangular array
     DRAM accesses an entire row
     Burst mode: supply successive words from a row with reduced latency (SDRAM)

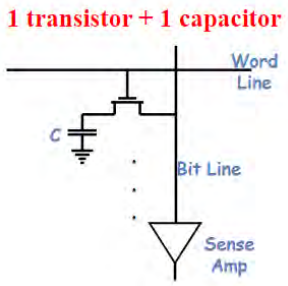


Figure 63: DRAM

   - Double data rate (DDR) DRAM
     Transfer on rising and falling clock edges
   - Quad data rate (QDR) DRAM
     Separate DDR inputs and outputs

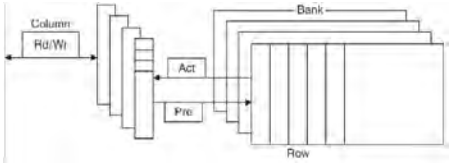


Figure 64: Advanced DRAM Organization

3) Flash Storage (Nonvolatile semiconductor storage)
   - 100x –1000x faster than disk
   - Smaller, lower power, more robust
   - But more $/GB than disk (between disk and-DRAM)

   Flash Types

   - NOR flash: bit cell like a NOR gate
     Random read/write access
     Used for instruction memory in embedded systems
   - NAND flash: bit cell like a NAND gate
     Denser (bits/area), but block-at-a-time access
     Cheaper per GB
     Used for USB keys, media storage, ...
   - Flash bits wears out after 10000's of accesses
     Not suitable for direct RAM or disk replacement
     Wear leveling: remap data to less used blocks

4) Disk Storage (Nonvolatile, rotating magnetic storage)
   Disk Sectors and Access

   - Each sector records
     Sector ID
     Data (512 bytes, 4096 bytes proposed)
     Error correcting code (ECC): Used to hide defects and recording errors

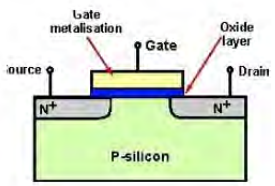


Figure 65: Flash Storage

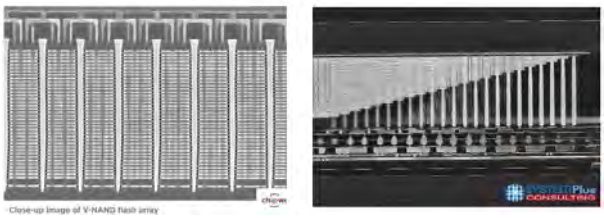


Figure 66: 3D flash makes SSD low cost and faster

Figure 67: Disk Storage

- Access to a sector involves
  Queuing delay if other accesses are pending
  Seek: move the heads
  Rotational latency
  Data transfer
  Controller overhead

rpm(round per minute)

2. Memory Hierarchy Introduction

1) Temporal locality: Items accessed recently are likely to be accessed again soon.

2) Spatial locality: Items near those accessed recently are likely to be accessed soon

2.1 Taking Advantage of Locality

1) Memory hierarchy

2) Store everything on disk

3) Copy recently accessed (and nearby) items from disk to smaller DRAM memory
   Main memory

4) Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory
   Cache memory attached to CPU

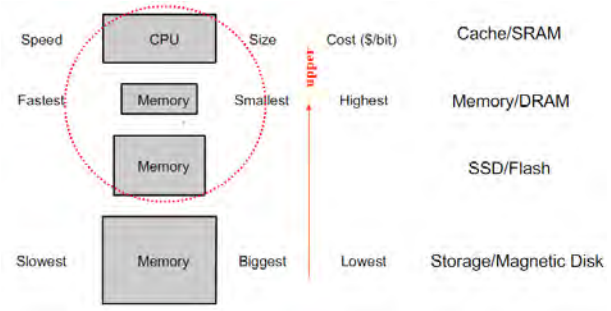


Figure 68: Memory Hierarchy Levels

2.2 Some important items

1) Hit: The CPU accesses the upper level and succeeds.
   Hit ratio: hits/accesses

2) Miss: The CPU accesses the upper level and fails.
   Time taken: miss penalty
   Miss ratio: misses/accesses= 1–hit ratio

3) Hit time: The time to access the upper level of the memory hierarchy, which includes the time needed to determine whether the access is a hit or a miss.

4) miss penalty: The time to replace a block in the upper level with the corresponding block from the lower level, plus the time to deliver this block to the processor.

2.3 Exploiting Memory Hierarchy

The method

1) Hierarchies bases on memories of different speeds and size

2) The more closely CPU the level is,the faster the one is.

3) The more closely CPU the level is,the smaller the one is.

4) The more closely CPU the level is,the more expensive.

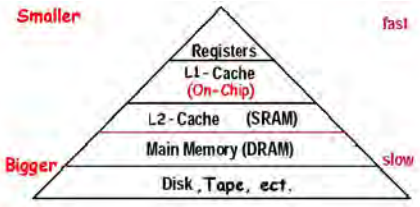


Figure 69: Levels in the Memory Hierarchy

3. The basics of Cache

3.1 Simple implementations

For each item of data at the lower level, there is exactly one location in the cache where it might be.

Two issues:

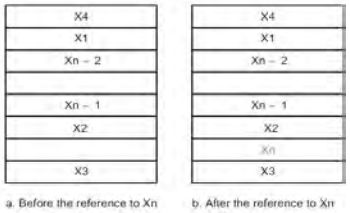


Figure 70: Simple implementations

- How do we know if a data item is in the cache?
- If it is, how do we find it?

Our first example: "direct mapped".

3.2 Direct Mapped Cache

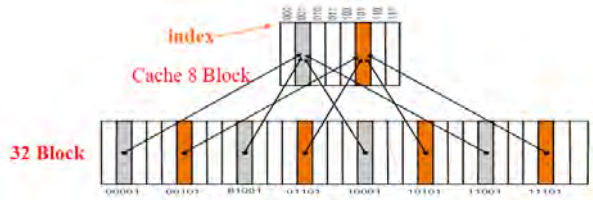


Figure 71: Direct Mapped Cache

Direct-mapping algorithm: (Block address) modulo (Number of blocks in the cache).

Fortunately, while the cache has 2^n blocks, the corresponding index is equal to the lowest n bits of memory block address. In **Figure** 71, $n = 3$.

3.2.1 Tags and Valid Bits To know which particular block is stored in a cache location:

1) Store block address as well as the data
2) Actually, only need the high-order bits
3) Called the tag

If there is no data in a location:

- Valid bit: 1 = present, 0 = not present
- Initially 0

3.2.2 Accessing a cache Memory block address is larger than cache block address.

3.2.3 Direct Mapped Cache Construction

- Tag bits: $32 - (n + m + 2)$
- Cache size: 2^n
- Index to reference words in block: m
- Total cache size: $2^n \times$ (block size + tag size + valid bit)

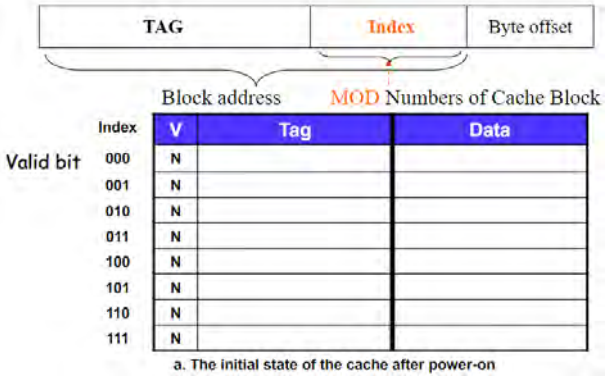


Figure 72: Accessing a cache

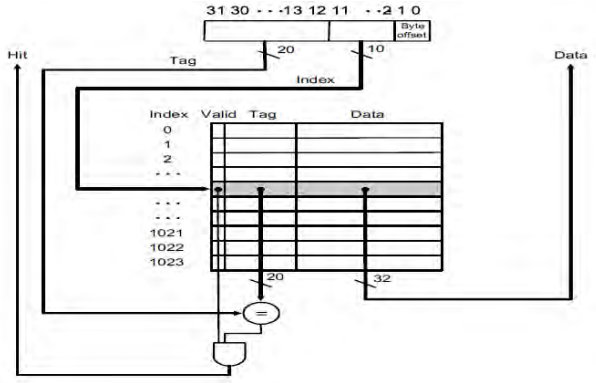


Figure 73: Direct Mapped Cache Construction

3.2.4 Bits in Cache Example:

How many total bits are required for a direct-mapped cache 16KB of data and 4-word blocks, assuming a 32-bit address?

Answer:

$$
\begin{aligned}
1\text{word} &= 32\text{bits} = 4\text{bytes} \\
16KiB &= 4K\text{words} = 2^{12}\text{words} \\
1\text{block} &= 4\text{words} = 16\text{bytes} \\
\text{index bits} &= \frac{2^{12}}{2^2} = 2^{10}\text{blocks} \\
\text{Tag bits} &= \text{address} - \text{index} - \text{block size} - \text{byte offset} \\
&= 32 - 10 - 2 - 2 = 18\text{bits} \\
\text{Valid bit} &= 1\text{bit} \\
\text{Total Cache size} &= 2^{10}(16 * 8 + 18 + 1) = 147K\text{bits}
\end{aligned}
$$

It is about $\frac{147}{128} \approx 1.15$ times as many as needed just for the data.

3.2.5 Handling Cache reads hit and Misses

Read hits

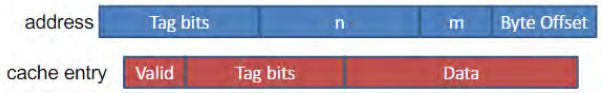


Figure 74: Bits in Cache

Read misses—two kinds of misses

- instruction cache miss
  When an instruction cache miss:
  1) Stall the CPU,
  2) fetch block from memory,
  3) deliver to cache,
  4) restart CPU read
- data cache miss

Write hits: Difference Strategy

- write-back: Cause Inconsistent
  Wrote the data into only the data cache
  Strategy — write back data from the cache to memory later
  Fast!
- write-through: Ensuring Consistent
  Write the data into both the memory the cache
  Strategy — writes always update both the cache and the memory
  Slower! — can use write buffer to speed up

Write misses: read the entire block into the cache, then write the word.

3.3 Four Questions for Memory Hierarchy Designers

Caching is a general concept used in processors, operating systems, file systems, and applications.

1) Q1: Where can a block be placed in the upper level? (Block placement)
   Fully Associative, Set Associative, Direct Mapped
2) Q2: How is a block found if it is in the upper level? (Block identification)
   Tag/Block
3) Q3: Which block should be replaced on a miss? (Block replacement)
   Random, LRU,FIFO
4) Q4: What happens on a write? (Write strategy)
   Write Back or Write Through (with Write Buffer)

3.3.1 Q1: Block Placement

- Direct mapped: Usually address MOD Number of blocks in cache
- Fully associative: Block can go anywhere in cache
- Set associative: Block can go in one of a set of places in the cache

Note that direct mapped is the same as 1-way set associative, and fully associative is m-way set-associative (for a cache with m blocks).

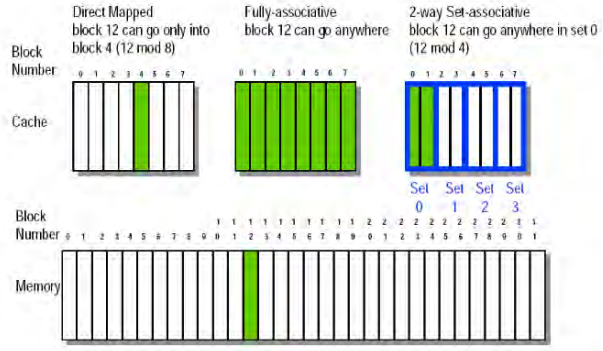


Figure 75: 8-32 Block Placement

3.3.2 Q2: Block Identification

- Tag: As a part of the address differentiation block
- Valid bit: tells if the contents of the cache block are valid

The Format of the Physical Address:

- Tag
- Index
  The set, in case of a set-associative cache
  The block, in case of a direct-mapped cache
- Block Offset: used to search for the word in 1 block, in case of a set-associative cache.
- Byte Offset: used to search for the bytes in 1 word.

address size = Tag size + index size + block offset + byte offset

3.3.3 Q3: Block Replacement

- In a direct-mapped cache, there is only one block that can be replaced
- In set-associative and fully-associative caches, there are N blocks (where N is the degree of associativity)

Strategy of Block Replacement:

1) Random replacement: randomly pick any block
2) Least-recently used (LRU): pick the block in the set which was least recently accessed

3) First in,first out(FIFO): Choose a block from the set which was first came into the cache

3.3.4 Q4: Write Strategy

- Write-through cache: the data is written to memory

  Advanced: Read misses don't result in writes, memory hierarchy is consistent and it is simple to implement.

- Write-back cache: the data is NOT written to memory

  Advantage: Writes occur at speed of cache and main memory bandwidth is smaller when multiple writes occur to the same block.

Write stall: When the CPU must wait for writes to complete during write through.

Write buffers: A small cache that can hold a few values waiting to go to main memory. This buffer helps when writes are clustered.

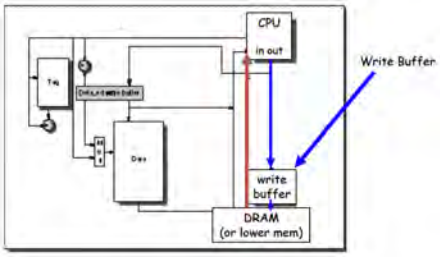


Figure 76: Write buffers

Write misses: If a miss occurs on a write (the block is not present), there are two options

1) Write allocate: The block is loaded into the cache on a miss before anything else occurs.

2) Write around (no write allocate): The block is only written to main memory. It is not stored in the cache.

In general, write-back caches use write-allocate , and write-through caches use write-around .

3.4 Designing the Memory system to Support Cache

Taking advantage of spatial locality to lower miss rates with many word in the block 77.

Make reading multiple words easier by using banks of memory 78.

3.4.1 Performance in different block size Increasing the block size tends to decrease miss rate 79.

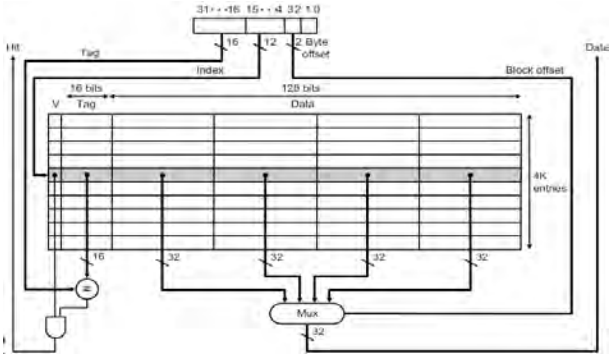


Figure 77: Larger blocks exploit spatial locality

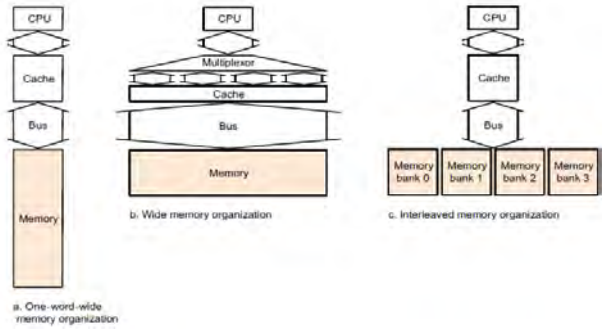


Figure 78: Different memory system

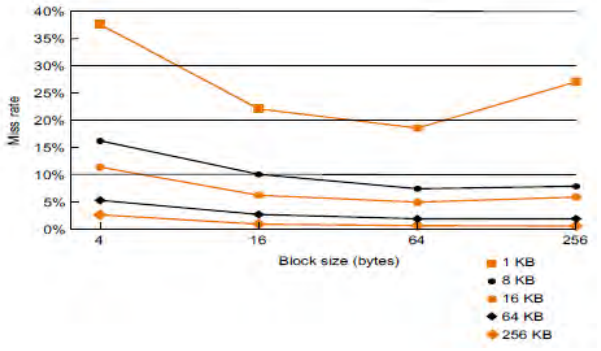


Figure 79: Performance in different block size

4. Measuring and improving cache performance

Discuss two questions:

1) How to measure cache performance?
2) How to improve performance?

The main contents are the following:

1) Measuring cache performance
2) Reducing cache misses by more flexible placement of blocks

3) Reducing the miss penalty using multilevel caches

$$\begin{aligned}&\text{Average Memory Assess Time (AMAT)}\\ &= \text{hit time} + \text{miss time}\\ &= \text{hit time} + \text{miss rate} \times \text{miss penalty}\end{aligned}$$

- Hit time: The time to access the upper level of the memory hierarchy, which includes the time needed to determine whether the access is a hit or a miss.
- Miss penalty: The time to replace a block in the upper level with the corresponding block from the lower level, plus the time to deliver this block to the processor.

4.1 Measuring cache performance

We use CPU time to measure cache performance. Before

$$\text{CPU}_{\text{time}} = \text{I} \times \text{CPI} \times \text{clock cycle time}$$

And now

$$\begin{aligned}&\text{CPU time}\\ &= \text{CPU execution clock cycles} + \text{Memory-stall clock cycles}\\ &\text{Memory-stall clock cycles}\\ &= \#\text{ of mem instructions} \times \text{miss ratio} \times \text{miss penalty}\\ &= \text{Read-stall cycles} + \text{Write-stall cycles}\end{aligned}$$

For Read-stall

$$\begin{aligned}&\text{Read-stall cycles}\\ &= \frac{\#\text{ of Read}}{\text{Program}} \times \text{Read miss rate} \times \text{Read miss penalty}\end{aligned}$$

For a write-through plus write buffer schemes:

$$\begin{aligned}&\text{Write-stall cycles}\\ &= \frac{\#\text{ of Write}}{\text{Program}} \times \text{Write miss rate} \times \text{Write miss penalty}\\ &+ \text{Write buffer stalls}\end{aligned}$$

Note that

- If the write buffer stalls are small, we can safely ignore them .
- If the cache block size is one word, the write miss penalty is 0.

4.1.1 Combine the reads and writes In most write-through cache organizations, the read and write miss penalties are the same, the time to fetch the block from memory. (这个表述不完全正确)

If we neglect the write buffer stalls, we get the following equation:

$$\begin{aligned}&\text{Memory-stall clock cycles}\\ &= \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}\end{aligned}$$

We can also write this as:

$$\begin{aligned}&\text{Memory-stall clock cycles}\\ &= \frac{\text{instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{instructions}} \times \text{Miss penalty}\end{aligned}$$

4.2 Calculating cache performance

Assume:

- instruction cache miss rate 2%
- data cache miss rate 4%
- CPI without any memory stalls 2
- miss penalty 100 cycles
- The frequency of all loads and stores in gcc is 36%

Question: How faster a processor would run with a perfect cache?

Answer:

$$\begin{aligned}\text{Instruction miss cycles} &= I \times 2\% \times 100 = 2.00I\\ \text{Data miss cycles} &= I \times 36\% \times 4\% \times 100 = 1.44I\\ \text{Total memory-satll cycles} &= 2.00I + 1.44I = 3.44I\end{aligned}$$

$$\begin{aligned}\text{CPI with stall} &= \text{CPI with perfect cache} + \text{Total memory-stalls}\\ &= (2 + 3.44)I = 5.44I\end{aligned}$$

$$\begin{aligned}\frac{\text{CPU time with stalls}}{\text{CPU time with perfect cache}} &= \frac{I \times \text{CPI}_{\text{stall}} \times \text{Clock cycle}}{I \times \text{CPI}_{\text{perfect}} \times \text{Clock cycle}}\\ &= \frac{\text{CPI}_{\text{stall}}}{\text{CPI}_{\text{perfect}}} = \frac{5.44}{2} = 2.72\end{aligned}$$

4.2.1 What happens if the processor is made faster Assume CPI reduces from 2 to 1.

$$\begin{aligned}\text{CPI with stall} &= \text{CPI with perfect cache} + \text{Total memory-stalls}\\ &= (1 + 3.44)I = 4.44I\end{aligned}$$

$$\frac{\text{CPU time with stalls}}{\text{CPU time with perfect cache}} = \frac{\text{CPI}_{\text{stall}}}{\text{CPI}_{\text{perfect}}} = \frac{4.44}{1} = 4.44$$

Ratio time for Memory stalls from $\frac{3.44}{5.44} = 63\%$ to $\frac{3.44}{4.44} = 77\%$

4.2.2 With Increased Clock Rate Suppose we increase the performance of the computer in the previous example by doubling its clock rate for same memory system.

Question : How much faster will the computer be with the faster clock to slow clock?

Answer:

$$\begin{aligned}\text{Total CPI} &= (2\% \times 200) + 36\%(4\% \times 200) = 6.88\\ \text{CPI with cache miss} &= 2 + 6.88 = 8.88\end{aligned}$$

$$\frac{\text{Performance with fast clock}}{\text{Performance with slow clock}} = \frac{\text{Execution time with slow clock}}{\text{Execution time with fast clock}}$$
$$= \frac{I \times \text{CPI}_{\text{slow clock}} \times \text{Clock cycle}}{I \times \text{CPI}_{\text{fast clock}} \times \text{Clock cycle}/2}$$
$$= \frac{5.44}{8.88/2} = 1.23$$

This, the computer with the faster clock is about 1.2 times faster rather than 2 time faster.

4.3 Solution: Reducing cache misses by more flexible placement of blocks

1) The disadvantage of a direct-mapped cache
2) The basics of a set-associative cache
3) Miss rate versus set-associative
4) Locating a block in the set-associative cache
5) Size of tags versus set associative
6) Choosing which block to replace

4.3.1 The basics of a set-associative cache Decreasing miss ratio with associativity.

A set-associative cacheis divided into some sets. A set contains several blocks. A memory block is mapped to a set in the cache. The mapping algorithm is:

Set number (Index) = Memory block number%Number of sets in the cache

- If a set has only one block, this set-associative cache is actually a direct-mapped cache.
- If a set-associative cache has only one set, this set-associative cache is called a fully-associative cache.

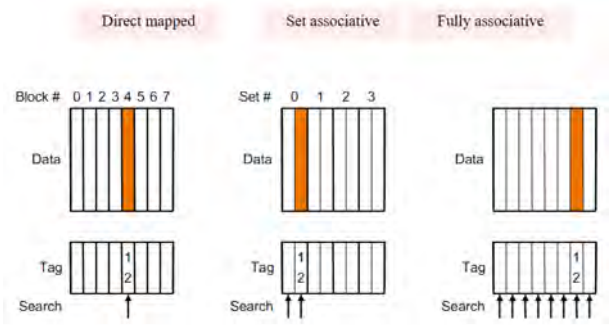


Figure 80: Memory block whose address is 12 in a cache with 8 blocks for different mapped

4.3.2 Locating a block in the set-associative cache The implementation of a four-way set-associative cache requires four comparators and a 4-to-1 multiplexor **Figure** 81.

4.3.3 Size of tags versus set associativity Assume:

- Cache size is 4K blocks
- Block size is 4 words

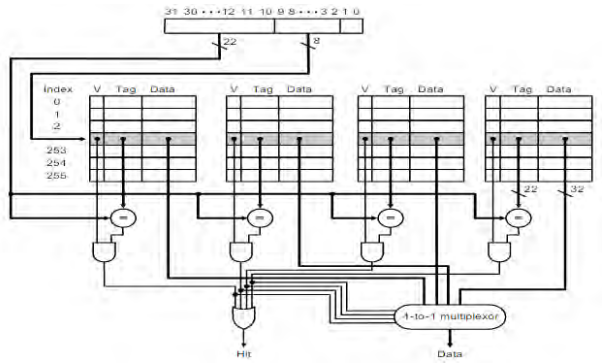


Figure 81: Locating a block in the set-associative cache

- Physical address is 32 bits

Question: Find the total number of set and total number of tag bits for variety associativity.

Answer:

$$\text{Offset size} = 4\text{word} \times 4\text{byte/word} = 16\text{byte} = 2^4$$

- For direct-mapped

$$\text{Number of cache block} = 2^{12}$$
$$\text{Bit of Tag} = (32 - 4 - 12) \times 4K = 64K\text{bits}$$

- For 2-way associative

$$\text{Number of cache block} = 2^{11}$$
$$\text{Bit of Tag} = (32 - 4 - 11) \times 4K = 67K\text{bits}$$

- For 4-way associative

$$\text{Number of cache block} = 2^{10}$$
$$\text{Bit of Tag} = (32 - 4 - 10) \times 4K = 72K\text{bits}$$

- For full associative

$$\text{Number of cache block} = 1$$
$$\text{Bit of Tag} = (32 - 4) \times 4K = 112K\text{bits}$$

4.3.4 Choosing which block to replace Least recently used (LRU): the block replaced is the one that has been unused for the longest time.

4.4 Decreasing miss penalty with multilevel caches

Add a second level cache:

- often primary cache is on the same chip as the processor
- use SRAMs to add another cache above primary memory (DRAM)
- miss penalty goes down if data is in 2nd level cache

Example:

- CPI of 1.0 on a 5GHz machine with a 2% miss rate, 100ns DRAM access
- Adding 2nd level cache with 5ns access time decreases miss rate to 0.5%

The CPI with one level of caching:

$$\begin{aligned}\text{Miss penalty to main memory} &= 100 \times 10^{-9}\text{s} \times 5 \times 10^{9}\text{Hz} \\ &= 500\text{clock cycles} \\ \text{Total CPI} &= 1 + \text{Memory-stall CPI} \\ &= 1 + 2\% \times 500 = 11\end{aligned}$$

The CPI with Two level of cache:

$$\begin{aligned}\text{Miss penalty with levels of cache} &= 5 \times 5 \\ &= 25\text{clock cycles} \\ \text{Total CPI} &= 1 + \text{Primary stalls CPI} \\ &\quad + \text{Secondary stalls CPI} \\ &= 1 + 2\% \times 25 + 0.5\% \times 500 = 4\end{aligned}$$

The processor with secondary cache is faster by $\frac{11}{4} = 2.8$

Using multilevel caches:

- try and optimize the hit time on the 1st level cache
- try and optimize the miss rate on the 2nd level cache

4.5 Miss Penalties (Include Write-back Cache)

Figure 82: Miss Penalties

If the write buffer stalls are small, we can safely ignore them. (No penalty on Write Memory)

5. Virtual Memory

Main Memory act as a "Cache" for the secondary storage. Motivation:

- Efficient and safe sharing of memory among multiple programs.
- Remove the programming burdens of a small, limited amount of main memory.

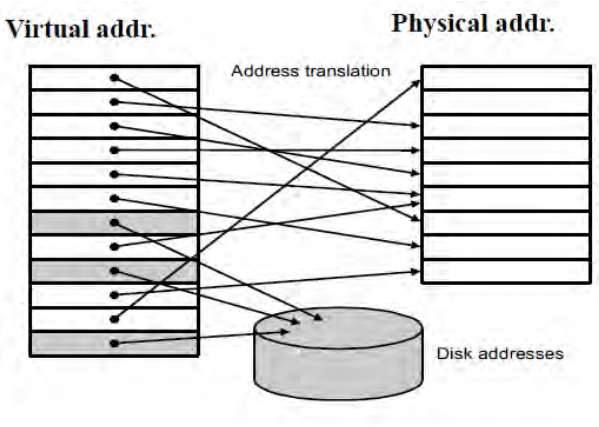


Figure 83: Main memory can act as a cache for the secondary storage (disk)

Translation of a program's address space to physical address.

Advantages:

- illusion of having more physical memory
- program relocation
- protection

5.1 Pages: virtual memory blocks

The number of virtual pages is larger than physical pages.

Page faults (miss): the data is not in memory, retrieve it from disk

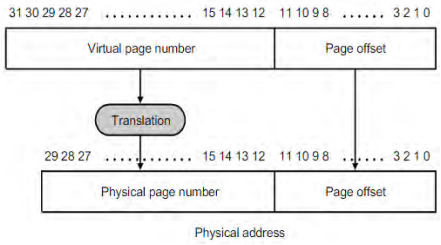


Figure 84: virtual memory blocks

5.2 Page Tables

1) Virtual to physical address.

2) Stored into the memory, indexed by the virtual page number

3) Each Entry in the table contains the physical page number for that virtual pages if the page is current in memory

4) Page table, Program counter and the page table register, specifies the state of the program. Each process has one page table.

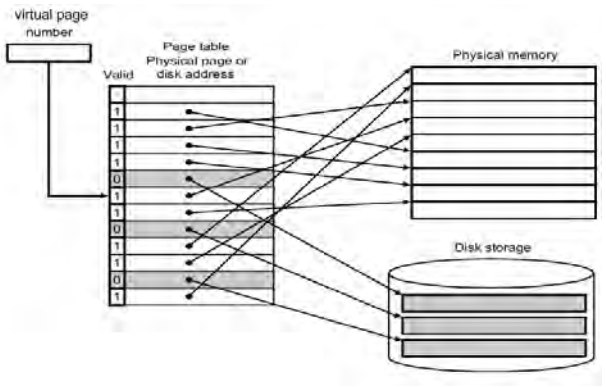


Figure 85: Page Tables

Virtual memory systems use fully associative mapping method.

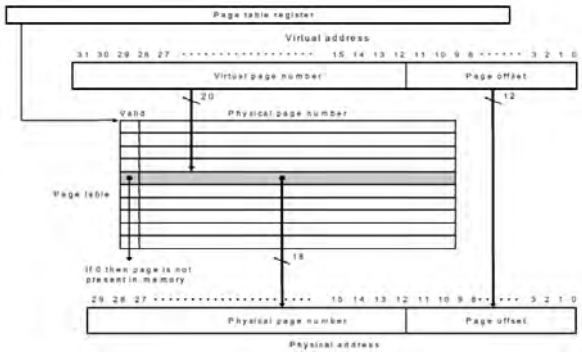


Figure 86: Page Table Diagram

5.3 Page Faults

If the valid bit for a virtual page is off, a page fault occur.

When a page fault occurs

1) The OS will find the page in the disk by the page table.
2) The OS will bring the requested page into main memory.

If all the pages in main memory are in use, the OS will use LRU strategy to choose a page to replace.

5.3.1 How large page table? Assume:

- Virtual address is 32 bits
- page size is 4KB
- Entry size is 4 bytes

$$\text{Number of page table entries} = \frac{2^{32}}{2^{12}} = 2^{20}$$

$$\text{Size of page table} = 2^{20} \times 2^2 = 4MB$$

5.4 Making Address Translation Fast–TLB

Virtual memory system must use write-back strategy.

5.4.1 Translation-lookaside Buffer(TLB) The TLB acts as Cache on the page table. A cache for address translations: translation look aside buffer.

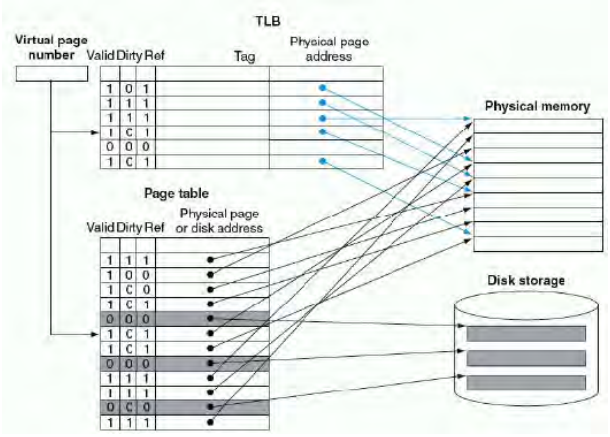


Figure 87: Translation-lookaside Buffer

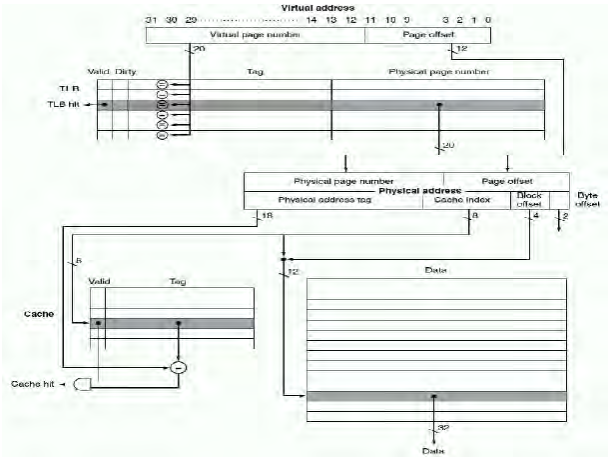


Figure 88: FastMATH Memory Hierarchy

6. Possible combinations of Event

Three different types of misses: TLB miss, page Fault, cache miss.

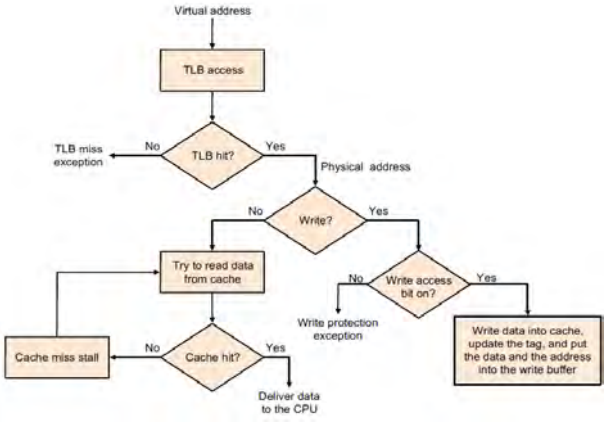


Figure 89: TLBs and caches

TLB	Page table	Cache	Possible? If so, under what circumstance?
hit	hit	miss	Possible, although the page table is never really checked if TLB hits.
miss	hit	hit	TLB misses, but entry found in page table; after retry, data is found in cache.
miss	hit	miss	TLB misses, but entry found in page table; after retry, data misses in cache.
miss	miss	miss	TLB misses and is followed by a page fault; after retry, data must miss in cache.
hit	miss	miss	Impossible: cannot have a translation in TLB if page is not present in memory.
hit	miss	hit	Impossible: cannot have a translation in TLB if page is not present in memory.
miss	miss	hit	Impossible: data cannot be allowed in cache if the page is not in memory.

Figure 90: Possible combinations of Event

VI Storage, Networks and Other Peripherals

1. Introduction

Performance of I/O system depends on:

- Connection between devices and the system
- The memory hierarchy
- The operating system

1.1 Typical I/O Devices

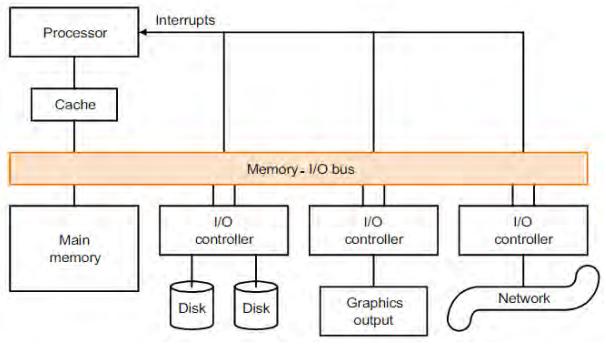


Figure 91: Typical IO Devices

1.2 Three Characters of I/O

1) Behavior: Input (read once), output (write only, cannot read), or storage (can be reread and usually rewritten)

2) Partner: Either a human or a machine is at the other end of the I/O device, either feeding data to input or reading data from output.

3) Data rate: The peak rate at which data are transferred between the I/O device and the main memory or processor.

1.3 I/O Performance Depends on the Application

- Throughput
- Response time
- Both throughput and response time

1.4 Amdahl's Law (阿姆达尔定律)

Sequential part can limit speedup. Remind us that ignoring I/O is dangerous.

2. Disk Storage and Dependability

- Magnetic disks: Hard disks
  (physically) larger
  higher storage volume
  reliable storage
  more than one platter
- SSD = Solid State Drive
  No mechanical part (all implemented in chip)
  $> 100X$ faster

2.1 The Organization of Hard Disk

- platters: disk consists of a collection of platters, each of which has two recordable disk surfaces
- tracks: each disk surface is divided into concentric circles
- sectors: each track is in turn divided into sectors, which is the smallest unit that can be read or written

2.2 To Access Data on Disk

- Seek: position read/write head over the proper track
- Rotational latency: wait for desired sector
- Transfer time: time to transfer a sector (1 KB/sector) of data
- Disk controller: control the transfer between the disk and the memory

2.3 Flash Storage

Nonvolatile solid-state storage.

Flash Types:

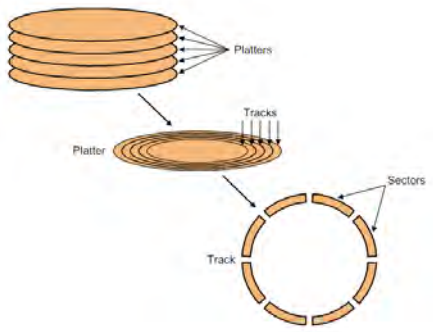


Figure 92: The Organization of Hard Disk

- NOR flash: bit cell like a NOR gate
- NAND flash: bit cell like a NAND gate

2.4 Dependability, Reliability, Availability

Computer system dependability is the quality of delivered service such that reliance can justifiably be placed on this service.

2.4.1 Measure

- MTTF (Mean Time to Failure)
- MTTR (Mean Time to Repair)
- MTBF (Mean Time Between Failures) = MTTF+ MTTR
- Availability

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

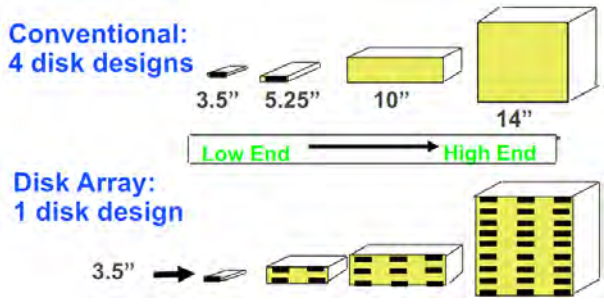


Figure 93: Use Arrays of Small Disks

2.4.2 Array Reliability

- Reliability of N disks = Reliability of 1 Disk/N
- AFR (annual failure rate) = percentage of devices to fail per year.

Reliability can be measured by "nines of availability" per year

One nine:	90%	=> 36.5 days of repair/year
Two nines:	99%	=> 3.65 days of repair/year
Three nines:	99.9%	=> 526 minutes of repair/year
Four nines:	99.99%	=> 52.6 minutes of repair/year
Five nines:	99.999%	=> 5.26 minutes of repair/year

Figure 94: Measure Reliability

2.4.3 Three Ways to Improve MTTF

1) Fault avoidance: preventing fault occurrence by construction

2) Fault tolerance: using redundancy to allow the service to comply with the service specification despite faults occurring, which applies primarily to hardware faults

3) Fault forecasting: predicting the presence and creation of faults, which applies to hardware and software faults

2.5 RAID: Redundant Arrays of (Inexpensive) Disks

A disk arrays replace larger disk

2.5.1 RAID 0: No Redundancy Data is striped across a disk array but there is no redundancy to tolerate disk failure.

2.5.2 RAID 1: Disk Mirroring/Shadowing Each disk is fully duplicated onto its "mirror".

Figure 95: RAID 1

2.5.3 RAID 3: Bit-Interleaved Parity Disk P contains sum of other disks per stripe mod 2 ("parity"). If disk fails, subtract P from sum of other disks to find missing information.

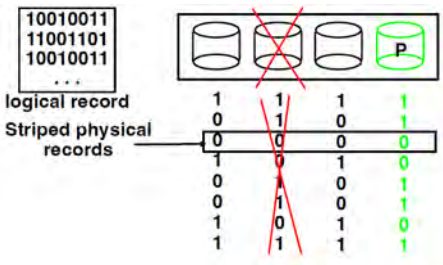


Figure 96: RAID 3

2.5.4 RAID 4:Block-Interleaved Parity Every sector has an error detection field. Rely on error detection field to catch errors on read, not on the parity disk.

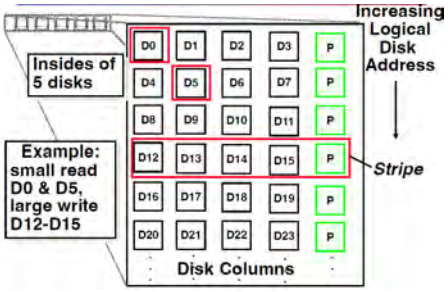


Figure 97: RAID 4

Problems of Disk Arrays: Small Writes

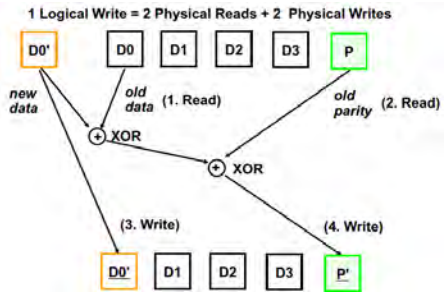


Figure 98: Small Write Algorithm

2.5.5 RAID 5: High I/O Rate Interleaved Parity Inspiration for RAID 5: can't parallel write P.

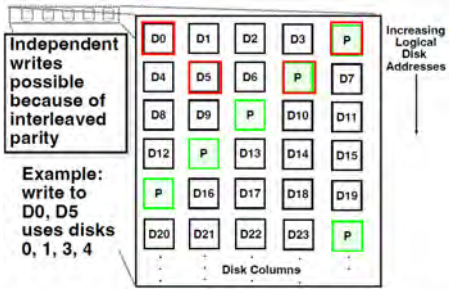


Figure 99: RAID 5

2.5.6 RAID 6: P+Q Redundancy When a single failure correction is not sufficient, Parity can be generalized to have a second calculation over data and anther check disk of information.

2.5.7 Summary: RAID Techniques

- Disk Mirroring, Shadowing (RAID 1)
- Parity Data Bandwidth Array (RAID 3)
- High I/O Rate Parity Array (RAID 5)

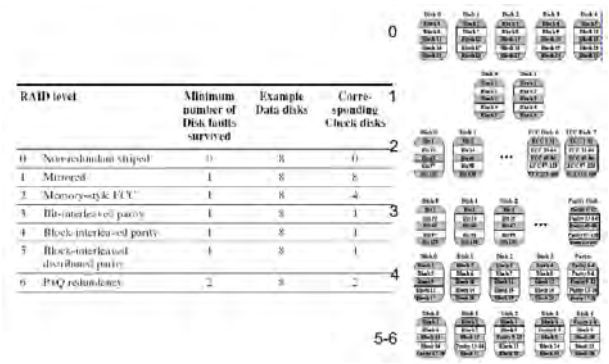


Figure 100: RAID

3. Buses and Connections between Processors Memory and I/O Devices

Buses and Other Connections

3.1 Bus Basics

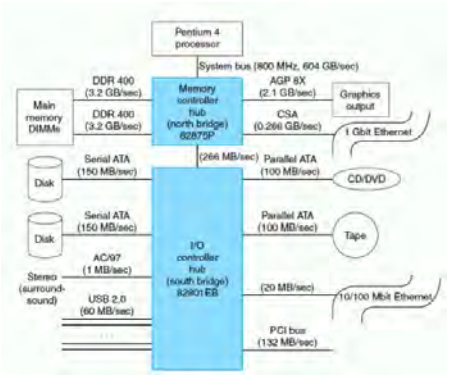


Figure 101: Bus

Bus: Shared communication link (one or more wires) A bus contains two types of lines:

1) Control lines: signal requests and acknowledgments, and to indicate what types of information is on the data lines.

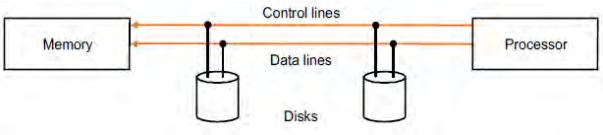
2) Data lines: carry information (e.g., data, addresses, and complex commands) between the source and the destination.

Bus transaction include two parts: sending the address and receiving or sending the data, which has two operations:

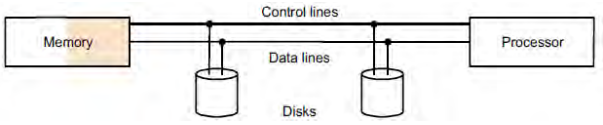
1) input: inputting data from the device to memory
2) output: outputting data to a device from memory

3.1.1 Output Operation

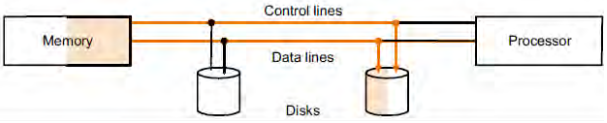
3.1.2 Input Operation



(a) Initial a read for memory. Control lines signal a read request to memory, while the data lines contain the address.

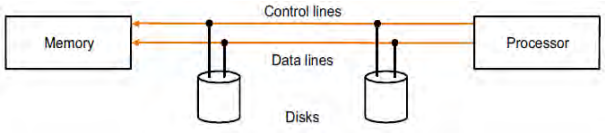


(b) Memory access the data.

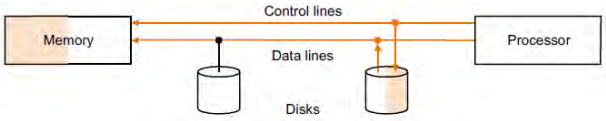


(c) Memory transfers data and signal data is available. The device stores data as it appears on the bus.

Figure 102: Output Operation



(a) Control lines indicate a write request for memory, while the data lines contain the address.



(b) When the memory is ready, it signals the device, which then transfers the data. The memory will store the data as it receives it . The device need not wait for the store to be completed.

Figure 103: Input Operation

3.1.3 Types of Buses

- Processor-memory : short high speed, custom design
- Backplane : high speed, often standardized
- I/O : lengthy, different devices, standardized

3.2 Synchronous vs. Asynchronous

- Synchronous bus: use a clock and a fixed protocol, fast and small but every device must operate at same rate and clock skew requires the bus to be short
- Asynchronous bus: don't use a clock and instead use handshaking
- Handshaking protocol: A serial of steps used to coordinate asynchronous bus transfers.

3.2.1 Asynchronous example (握手协议)

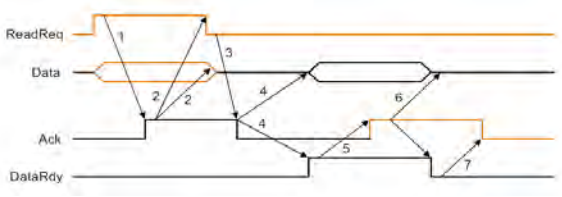


Figure 104: Asynchronous example

1) When memory sees the ReadReq line, it reads the address from the data bus, begin the memory read operation, then raises Ack to tell the device that the ReadReq signal has been seen.

2) I/O device sees the Ack line high and releases the ReadReq data lines.

3) Memory sees that ReadReq is low and drops the Ack line.

4) When the memory has the data ready, it places the data on the data lines and raises DataRdy.

5) The I/O device sees DataRdy, reads the data from the bus , and signals that it has the data by raising ACK.

6) The memory sees Ack signals, drops DataRdy, and releases the data lines.

7) Finally, the I/O device, seeing DataRdy go low, drops the ACK line, which indicates that the transmission is completed.

3.3 Bus Arbitration

Obtaining Access to the Bus. Deciding which bus master gets to use the bus next.

In a bus arbitration scheme, a device wanting to use the bus signals a bus request and is later granted the bus. Four bus arbitration schemes:

1) daisy chain arbitration (not very fair)
2) centralized, parallel arbitration (requires an arbiter), e.g., PCI
3) self selection, e.g., NuBus used in Macintosh
4) collision detection, e.g., Ethernet

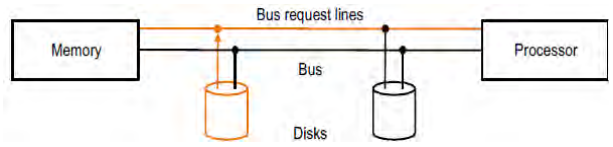
Two factors in choosing which device to grant the bus:

1) bus priority
2) fairness

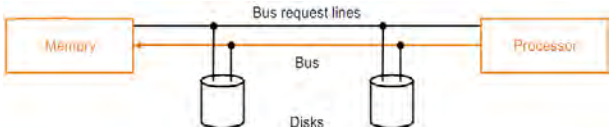
4. Interfacing I/O Devices to the Memory, Processor, and Operating System

Three characteristics of I/O systems:

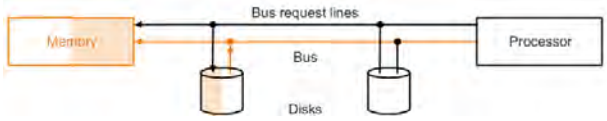
1) **shared** by multiple programs using the processor.



(a) First, the device generates a bus request to inform the processor that it wants to use the bus.



(b) The processor responds and generates appropriate bus control signals. For example, if the devices wants to perform output from memory, the processor asserts the read request lines to memory.



(c) The processor also notifies the device that its bus request is being processed; as a result, the device knows it can use the bus and places the address for the request on the bus.

Figure 105: Bus Arbitration

2) often use **interrupts** to communicate information about I/O operations.

3) The low-level control of an I/O devices is **complex**

Three types of communication are required:

1) The OS must be able to give **commands** to the I/O devices.

2) The device must be able to **notify** the OS, when I/O device **completed** an operation or has encountered an **error**.

3) Data must be transferred between memory and an I/O device

4.1 Giving Commands to I/O Devices

Two methods used to address the device:

- memory-mapped I/O: portions of the memory address space are assigned to I/O devices, and lw and sw instructions can be used to access the I/O port.
- special I/O instructions

4.2 Communication with the Processor

- Polling: The processor periodically checks status bit to see if it is time for the next I/O operation.
- Interrupt: When an I/O device wants to notify processor that it has completed some operation or needs attentions, it causes processor to be interrupted.
- DMA (direct memory access): the device controller transfer data directly to or from memory without involving processor.

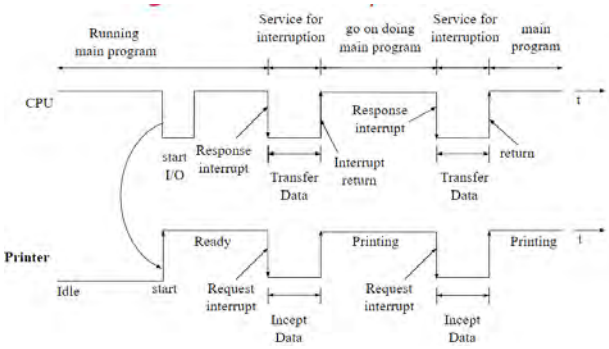


Figure 106: Interrupt-Driven I/O Mode

4.2.1 DMA Transfer Mode A DMA transfer need three steps:

1) The processor sets up the DMA by supplying some information, including the identity of the device, the operation, the memory address that is the source or destination of the data to be transferred, and the number of bytes to transfer.

2) The DMA starts the operation on the device and arbitrates for the bus. If the request requires more than one transfer on the bus, the DMA unit generates the next memory address and initiates the next transfer.

3) Once the DMA transfer is complete, the controller interrupts the processor, which then examines whether errors occur.

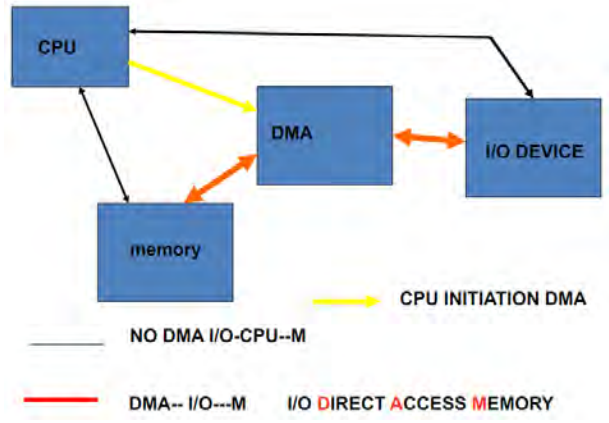


Figure 107: DMA Transfer Mode

4.2.2 Compare Polling, Interrupts, DMA The disadvantage of polling is that it wastes a lot of processor time. When the CPU polls the I/O devices periodically, the I/O devices maybe have no request or have not get ready. If the I/O operations is interrupt driven, the OS can work on other tasks while data is being read from or written

to the device. Because DMA doesn't need the control of processor, it will not consume much of processor time.