1 - Introduction

1.1 Programming Language

- Programming language: a formal language that specifies a set of instructions for a computer to implement specific algorithms to solve problems

High-level program
Eg: C, Java, Python, ECMAScript

```
int i, a = 0;
for (i=1; i<=10; i++) {
        a = a + i*i;
}
```

```
a = 0
for i in range(1,11):
  a = a + i*i
```

Low-level program
Eg: MIPS (IT5002)

```
      addi $t1, $zero, 10
      add  $t1, $t1, $t1
      addi $t2, $zero, 10
Loop: addi $t2, $t2, 10
      addi $t1, $t1, -1
      beq  $t1, $zero, Loop
```

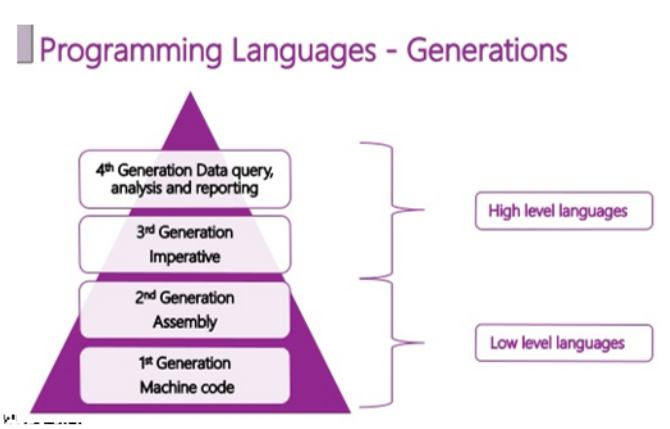
Machine code

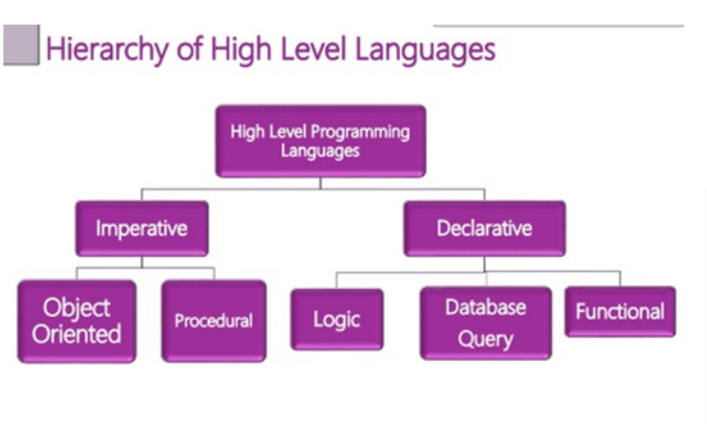
```
00100000000010010000000000001010
00000001001010010100100000100000
. . .
```

Computers can execute only machine code directly.

- 1st Generation languages:
  - Machine language
  - Directly executable by machine
  - Machine dependent
  - Efficient code but difficult to write
- 2nd generation languages:
  - Assembly language
  - Need to be **translated(assembled)** into machine code for execution
  - Efficient code, easier to write than machine code
- 3rd generation language:
  - Closer to English
  - Need to be **translated (complied or interpreted)** into machine code for execution
  - Example: FORTRAN, COBOL, C, BASIC
- 4th generation language:
  - Require fewer instructions than 3GL
  - Used with databases (query languages, report generators, forms designers)

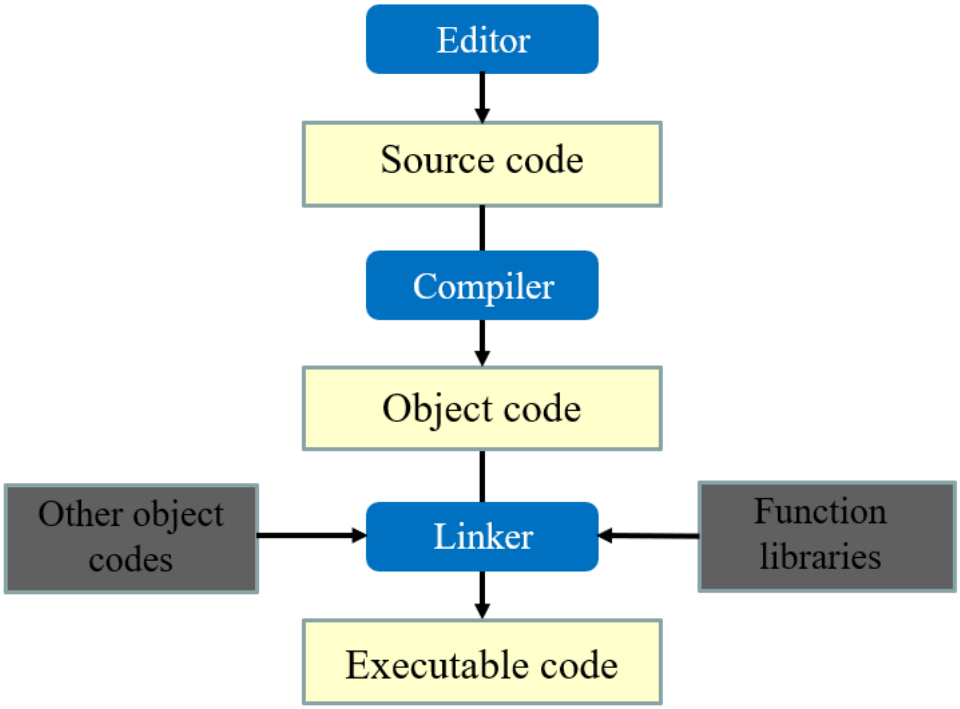
    - Example: SQL, PostScript, Mathematica
- 5th generation language:
    - Used mainly AI research
    - Declarative languages
    - Functional languages (Lisp, Scheme, SML)
    - Logic programming (Prolog)
- "Generational" classification of high level languages (3GL and later) was never fully precise
- A different classification is based on paradigm





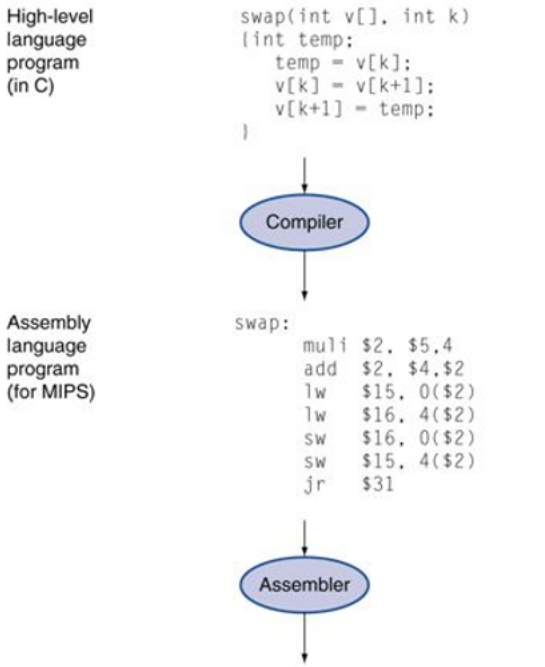
1.1.1 C Programming Language

- C is an **imperative procedural language (命令式程序语言)**
- C provides constructs that map efficiently to typical machine instructions
- C is a high-level language very close to the machine level, hence sometimes it is called "mid-level"

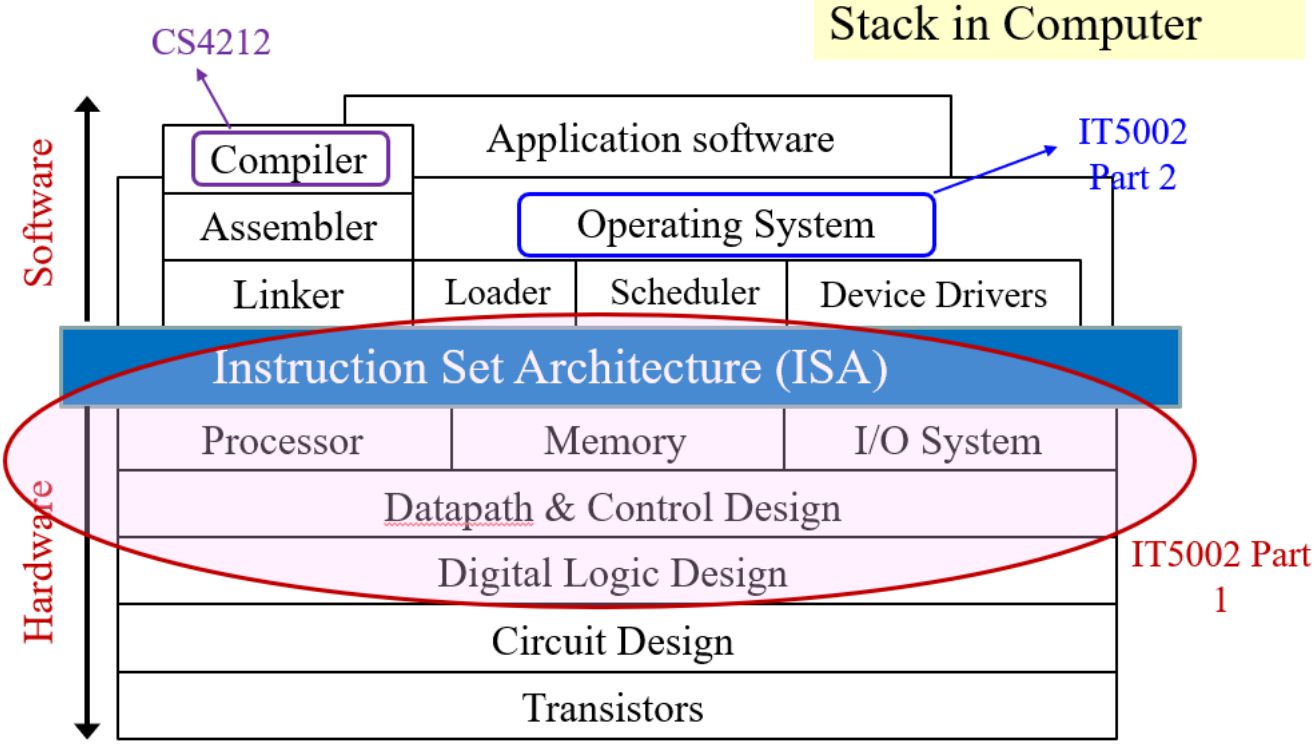


1.2 Abstraction

- High-level language
  - Level of abstraction closer to problem domain
  - Provides productivity and portability
- Assembly language
  - Textual and symbolic representation of instructions
- Machine code (object code or binary)
  - Binary bits of instructions and data



Hardware/Software Stack in Computer



Level of Representation

High Level Language Program (e.g., C)

Compiler

Assembly Language Program (e.g.,MIPS)

Assembler

Machine Language Program (MIPS)

Machine Interpretation

Hardware Architecture Description (Logic, Logisim, etc.)

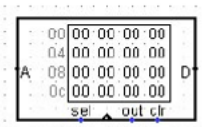
Architecture Implementation

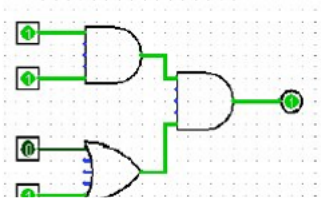
Logic Circuit Description (Logisim, etc.)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw   $t0, 0($2)
lw   $t1, 4($2)
sw   $t1, 0($2)
sw   $t0, 4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

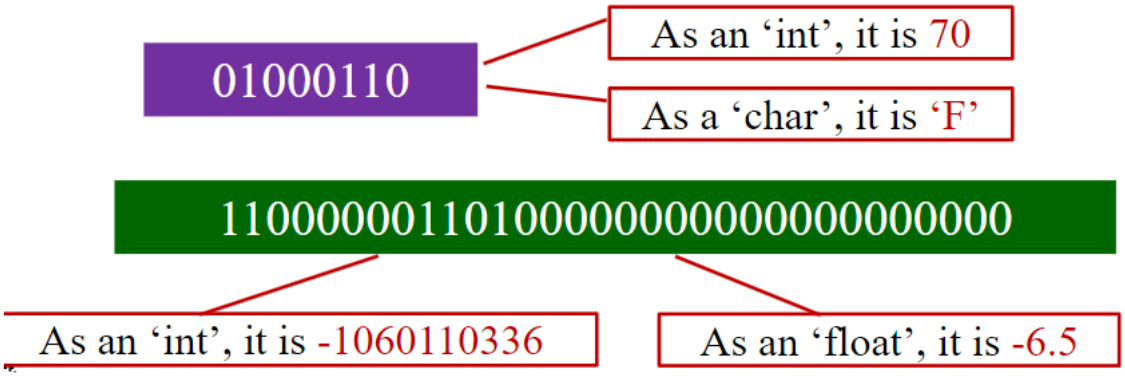




2 - Number Systems

2.1 Data Representation

- Basic data types in C:
  - `int`, with variants `short`, `long`
  - `float`
  - `double`
  - `char`
- How data is represented depends on its type:



- Data are internally represented as sequence of bits (binary digits). A bit is either 0 or 1
- Other units:
  - Byte = 8 bits
  - Nibble = 4 bits
  - Word = Multiple of bytes (eg: 1 byte, 2 bytes, 4 bytes, etc) depending on the computer architecture
- N bits can represent up to 2^n values
  - 2 bits represent up to 4 values (00, 01, 10, 11)
- To represent M values, $\lceil log_2 M \rceil$ bits required
  - 32 values requires 5 bits; 1000 values require 10 bits

2.2 Decimal (base 10) Number System

- A weighted-positional number system
- Base (also called radix) is 10
- Symbols/digits = $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$
- Each position has a weight of power of 10
  - $(7594.36)_{10} = (7 \times 10^3) + (5 \times 10^2) + (9 \times 10^1) + (4 \times 10^0) + (3 \times 10^{-1}) + (6 \times$

2.3 Other Number Systems

- In some programming languages/software, special notations are used to represent numbers in certain bases
  - In C
    - prefix 0 for octal. Eg: `032` represents the octal number $(32)_8$
    - prefix 0x for hexadecimal. Eg: `0x32` represents the hexadecimal number $(32)_{16}$
  - In QTSpim (a MIPS simulator)
    - prefix 0x for hexadecimal.
  - In Verilog, the following values are the same
    - `8'b11110000`: an 8-bit binary value 11110000
    - `8'hF0`: an 8-bit binary value represented in hexadecimal F0
    - `8'd240`: an 8-bit binary value represented in decimal 240

2.4 Base-R to Decimal Conversion

- $1101.101_2 = 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-3} = 8 + 4 + 1 + 0.5 + 0.12$
- $572.6_8 = 5 \times 8^2 + 7 \times 8^1 + 2 \times 8^0 + 6 \times 8^{-1} = 320 + 56 + 2 + 0.75 = 378.75_8$
- $2A.8_{16} = 2 \times 16^1 + 10 \times 16^0 + 8 \times 16^{-1} = 32 + 10 + 0.5 = 42.5_{10}$

2.5 Decimal to Binary Conversion

- For whole numbers
  - Repeated Division-by-2 method
- For fractions
  - Repeated Multiplication-by-2 method

2.5.1 Repeated Division-by-2

Repeated Divide

- To convert a whole number to binary, use **successive division by 2** until the quotient is 0. The remainders from the answer, with the first remainder as the Least Significant Bit (LSB) and the last as the Most Significant Bit (MSB)

$(43)_{10} = (\ 101011\ )_2$

2	43	
2	21 rem 1	← LSB
2	10 rem 1	
2	5 rem 0	
2	2 rem 1	
2	1 rem 0	
	0 rem 1	← MSB

Repeated Division-by-2: 这种方法经常被用于将十进制数转换为二进制数。

步骤如下：

1. 用2除以给定的十进制数，记录商和余数。
2. 再用2除以上一步得到的商，再次记录商和余数。
3. 重复上述步骤，直到商变为0为止。
4. 从上到下读取余数，就得到了对应的二进制表示。

例如，将十进制数13转换为二进制：

```
Copy code13 ÷ 2 = 6 商 1 余数
6 ÷ 2 = 3 商 0 余数
3 ÷ 2 = 1 商 1 余数
1 ÷ 2 = 0 商 1 余数

从上到下读取余数得到：1101，所以13的二进制表示是1101。
```

2.5.2 Repeated Multiplication-by-2

Repeated Multiply

- To convert **decimal fractions** to binary, **repeated multiplication by 2** is used, until the fractional product is 0 (or until the desired number of decimal places). The carried digits, or *carries*, produce the answer, with the first carry as the MSB, and the last as the LSB.

$(0.3125)_{10} = (\ .0101\ )_2$

	Carry	
$0.3125\times2=0.625$	0	←MSB
$0.625\times2=1.25$	1	
$0.25\times2=0.50$	0	
$0.5\times2=1.00$	1	←LSB

Repeated Multiplication-by-2: 这种方法经常被用于将二进制小数转换为十进制小数。

步骤如下：

1. 将给定的二进制小数的最高位（最左边的位）乘以2。
2. 记录该乘积的整数部分。
3. 将乘积的小数部分再乘以2。
4. 重复上述步骤，直到得到的小数部分为0或达到所需的精度。

例如，将二进制小数0.101转换为十进制：

```
rustCopy code0.101 × 2 = 1.01  → 记录整数部分 1
0.01 × 2  = 0.02  → 记录整数部分 0
0.02 × 2  = 0.04  → 记录整数部分 0（如果需要更多精度则继续，否则可以停止）

从上到下读取整数部分得到：100，表示二进制的0.101等于十进制的0.5。
```

2.6 Conversion Between Decimal and Other Bases

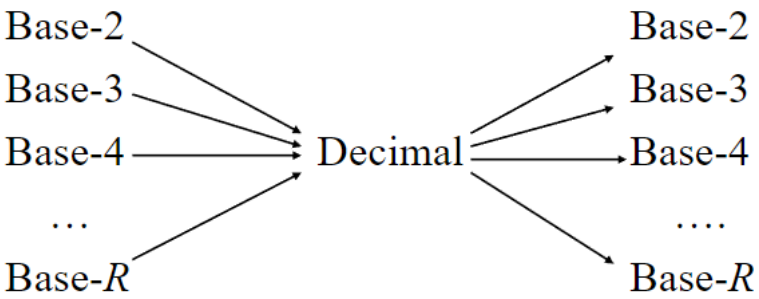
- **Base-R to decimal:** multiply digits with their corresponding weights
- **Decimal to binary (base 2)**
  - For whole numbers: repeated division-by-2 (2.5.1)
  - For fraction numbers: repeated multiplication-by-2 (2.5.2)
- **Decimal to base-R**
  - For whole numbers: repeated division-by-R
  - For fraction numbers: repeated multiplication-by-R

总结：

不管是什么进制，均可使用2.5章内使用的方法，将除以2或乘以2替换进制数字

2.7 Conversion Between Bases

- In general, conversion between bases can be done via decimal:



2.8 Binary to Octal/Hexadecimal Conversion

- Binary → Octal: partition in groups of 3
  - $(10\ \ 111\ \ 011\ \ 001\ .\ 101\ \ 110)_2 = (2731.56)_8$
- Octal → Binary: reverse
  - $(2731.56)_8 = (10\ \ 111\ \ 011\ \ 001\ .\ 101\ \ 110)_2$
- Binary → Hexadecimal: partition in groups of 4
  - $(101\ \ 1101\ \ 1001\ .\ 1011\ \ 1000)_2 = (5D9.B8)_{16}$
- Hexadecimal → Binary: reverse
  - $(5D9.B8)_{16} = (101\ \ 1101\ \ 1001\ .\ 1011\ \ 1000)_2$

2.9 ASCII Code

- **ASCII code** and **Unicode** are used to represent characters `('a', 'C', '?', '\0')`
- ASCII
  - American Standard Code for Information Interchange
  - 7 bits, plus 1 parity bit (odd or even parity)

Character	ASCII Code
0	0110000
1	0110001
...	...
9	0111001
:	0111010
A	1000001
B	1000010
...	...
Z	1011010
[	1011011
\	1011100

'A': 1000001 (or 65_{10})

	MSBs							
LSBs	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC_1	!	1	A	Q	a	q
0010	STX	DC_2	"	2	B	R	b	r
0011	ETX	DC_3	#	3	C	S	c	s
0100	EOT	DC_4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	O	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

- Integers (0 to 127) and characters are 'somewhat' interchangeable in C

01000110

As an 'int', it is 70

As a 'char', it is 'F'

CharAndInt.c

```
int num = 65;
char ch = 'F';

printf("num (in %%d) = %d\n", num);
printf("num (in %%c) = %c\n", num);
printf("\n");

printf("ch (in %%c) = %c\n", ch);
printf("ch (in %%d) = %d\n", ch);
```

```
num (in %d) = 65
num (in %c) = A

ch (in %c) = F
ch (in %d) = 70
```

```c
int main() {
 int i, n = 2147483640;
 for (i=1; i⩽10; i++) {
     n = n + 1;
 }
 printf("n = %d\n", n);
}
```

对于这一段代码，其输出是什么?

这段代码中， `int` 数据类型的整数会溢出。在多数计算机系统中，一个 `int` 数据类型通常占据4个字节（32位），其范围是从 `-2,147,483,648` （即 `-2^31` ）到 `2,147,483,647` （即 `2^31 - 1` ）。

当 `n` 的值是 `2,147,483,640` ，并在循环中加了10次，它的值会变为 `2,147,483,650` 。这个值超出了 `int` 的最大值 `2,147,483,647` 。

因此，当加1到 `2,147,483,647` ，它会溢出并回绕到 `int` 的最小值 `-2,147,483,648` ，然后再从这个值开始增加。

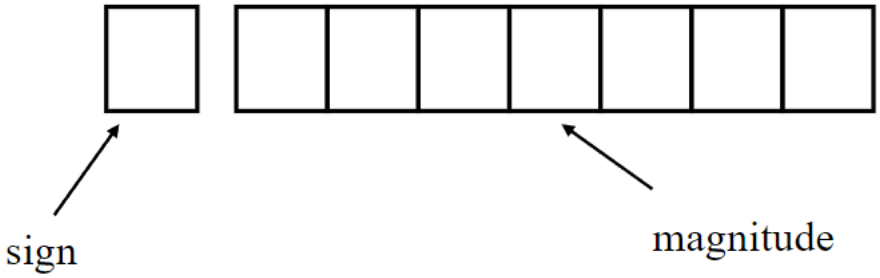
```
2,147,483,640 + 1 = 2,147,483,641
2,147,483,641 + 1 = 2,147,483,642
...
2,147,483,646 + 1 = 2,147,483,647  // 这是int的最大值
2,147,483,647 + 1 = -2,147,483,648 // 溢出，变成int的最小值
-2,147,483,648 + 1 = -2,147,483,647
-2,147,483,647 + 1 = -2,147,483,646
```

2.10 Negative Numbers

- Unsigned numbers: only non-negative values
- Signed numbers: include all values (positive and negative)
- There are 3 common representations for signed binary numbers:
  - Sign-and-magnitude
  - 1s Complement
  - 2s Complement

2.10.1 Sign-and-Magnitude

- The sign is represented by a ‘sign bit’
  - `0` for +
  - `1` for -
- For example: a 1-bit sign and 7-bit magnitude format



- Example:
  - `00110100` $\rightarrow +110100_2 = +52_{10}$
  - `10010011` $\rightarrow -10011_2 = -19_{10}$
- For 8-bit binary number:
  - Largest value: `01111111`, 127_{10}
  - Smallest value: `11111111`, -127_{10}
  - Zeros:
    - `00000000`, $+0_{10}$
    - `10000000`, -0_{10}
  - Range (for 8-bit): -127_{10} to $+127_{10}$
- For n-bit sign-and-magnitude representation, the range of values should be:
  - $-2^{n-1}+1$ to $2^{n-1}-1$
- Negate a number, just **invert the sign bit**
  - Examples:
    - Negate 00100001_2 (decimal 33)
      - 10100001_2 (decimal -33)
    - Negate 10000101_2 (decimal -5)
      - 00000101_2 (decimal 5)

2.10.2 1s Complement 一进制补码

- Given a number `x` which can be expressed as an n-bit binary number, its **negated value** can be obtained in **1's-complement** representation using:

$$-x = 2^n - x - 1$$

- Example: With an 8-bit number `00001100` (or 12_{10}), its negated value expressed in 1's-complement is:

$$\begin{aligned} -00001100_2 &= 2^8 - 12 - 1 \\ &= 243 \\ &= 11110011_{1s} \end{aligned}$$

- (This means that -12_{10} is written as `11110011` in 1s-complement representation)
- Technique to negate a value: **invert all the bits**
- Largest value: `0111 1111` = $+127_{10}$
- Smallest value: `1000 0000` = -127_{10}
- Zeros:
  - `0000 0000` = $+0_{10}$
  - `1111 1111` = -0_{10}
- Range (for 8 bits): -127_{10} to $+127_{10}$
- Range (for n bits): $-(2^{n-1}-1)$ to $2^{n-1}-1$
- The *most significant bit (MSB)* still represents the sign: 0 for positive, 1 for negative
- Examples:
  - $(14_{10}) = (00001110)_2 = (00001110)_{1s}$
  - $-(14)_{10} = -(00001110)_2 = (11110001)_{1s}$

> - 对于一个二进制数，它的1's complement是将该数中的每一位都取反。换句话说，将所有的0变为1，所有的1变为0。
> - 例如，考虑一个8位二进制数 `1101 0101` 。它的1's complement是 `0010 1010` 。

2.10.3 2s Complement

- Given a number x which can be expressed as an n-bit binary number, its *negated value* can be obtained in **2s-complement** representation using:

$$-x = 2^n - x$$

- Example: With an 8-bit number `00001100` (or 12_{10}), its negated value expressed in 2s-complement is:

$$\begin{aligned} -00001100_2 &= 2^8 - 12 \\ &= 244 \\ &= ((11110011)_{1s} + 1)_{2s} \\ &= 11110100_{2s} \end{aligned}$$

- This means that -12_{10} is written as `1111 0100` in 2s-complement representation
- Technique to negate a value: **invert all the bits, then add 1**
- Largest value: `0111 1111` = $+127_{10}$
- Smallest value: `1000 0000` = -128_{10}
- Zero: `0000 0000` = $+0_{10}$

- Range (for 8 bits): -128_{10} to $+127_{10}$
- Range (for n bits): $-(2^{n-1})$ to $2^{n-1}-1$
- The *most significant bit (MSB)* still represents the sign: 0 for positive, 1 for negative
- Examples:
  - $(14)_{10} = (00001110)_2 = (00001110)_{2s}$
  - $-(14)_{10} = -(00001110)_2 = (11110010)_{2s}$

2.10.4 Comparisons

4-bit system

Positive values

Value	Sign-and-Magnitude	1s Comp.	2s Comp.
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000

Negative values

Value	Sign-and-Magnitude	1s Comp.	2s Comp.
-0	1000	1111	-
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8	-	-	1000

2.10.5 Complement on Fractions

- We can extend the idea of complement on fractions
- Examples:
  - Negate `0101.01` in 1s-complement
    - Answer: `1010.10`
  - Negate `111000.101` in 1s-complement
    - Answer: `000111.010`
  - Negate `0101.01` in 2s-complement
    - Answer: `1010.11`

2.10.6 2s Complement Addition/Subtraction

- Algorithm for addition of integers, $A + B$:
  1. Perform binary addition on the two numbers
  2. Ignore the carry out of the MSB
  3. Check for overflow. Overflow occurs if the 'carry in' and 'carry out' of the MSB are different, or if result is opposite sign of A and B
- Algorithm for subtraction of integers, $A - B$:

$$A - B = A + (-B)$$

  1. Take 2s-complement of B
  2. Add the 2s-complement of B to A

Overflow

- Signed numbers are of a fixed range
- If the result of addition/subtraction goes beyond this range, an **overflow** occurs
- Overflow can be easily detected:
  - positive add positive → negative
  - negative add negative → positive
- Example: 4-bit 2s-complement system
  - Range of value: -8_{10} to 7_{10}
  - $0101_{2s} + 0110_{2s} = 1011_{2s}$
    $5_{10} + 6_{10} = -5_{10}$ (Overflow!)
  - $1001_{2s} + 1101_{2s} = 10110_{2s}\ (\text{discard end-carry}) = 0110_{2s}$
    $-7_{10} + -3_{10} = 6_{10}$ (Overflow!)

Examples: 4-bit system

```
  +3      0011
+ +4    + 0100
----    -------
  +7      0111
----    -------
```
No overflow

```
  -2      1110
+ -6    + 1010
----    -------
  -8     11000
----    -------
```
No overflow

```
  +6      0110
+ -3    + 1101
----    -------
  +3     10011
----    -------
```
No overflow

```
  +4      0100
+ -7    + 1001
----    -------
  -3      1101
----    -------
```
No overflow

```
  -3      1101
+ -6    + 1010
----    -------
  -9     10111
----    -------
```
Overflow!

```
  +5      0101
+ +6    + 0110
----    -------
 +11      1011
----    -------
```
Overflow!

Examples: 4-bit system

- 4 – 7
- Convert it to 4 + (-7)

```
  +4      0100
+ -7    + 1001
----    -------
  -3      1101
----    -------
```
No overflow

- 6 – 1
- Convert it to 6 + (-1)

```
  +6      0110
+ -1    + 1111
----    -------
  +5     10101
----    -------
```
No overflow

- -5 – 4
- Convert it to -5 + (-4)

```
  -5      1011
+ -4    + 1100
----    -------
  -9     10111
----    -------
```
Overflow!

在二进制的2's-complement加减法中，判断溢出的情况是基于加法的结果与两个操作数的关系来确定的。下面我将为您解释如何判断正溢出和负溢出。

1. **正溢出**:
   - 当两个正数相加得到一个负数结果时，就发生了正溢出。
   - 具体判断方式为：两个操作数的最高位（符号位）都是0，但结果的最高位是1。
2. **负溢出**:
   - 当两个负数相加得到一个正数结果时，就发生了负溢出。
   - 具体判断方式为：两个操作数的最高位都是1，但结果的最高位是0。

加减法的关系：减法可以看作加上一个数的2's complement。所以，如果你知道如何检测加法的溢出，你也可以应用这个知识来检测减法的溢出。

2.10.7 1s Complement Addition/Subtraction

- Algorithm for addition of integers, $A + B$:
  1. Perform binary addition on the two numbers
  2. If there is a carry out of the MSB, add 1 to the result
  3. Check for overflow. Overflow occurs if result is opposite sign of A and B
- Algorithm for subtraction of integers, $A - B$:

$$A - B = A + (-B)$$

  1. Take 1s-complement of B
  2. Add the 1s-complement of B to A

- Examples: 4-bit system

```
  +3      0011
+ +4    + 0100
----    -------
  +7      0111
----    -------
```
No overflow

```
  +5      0101
+ -5    + 1010
----    -------
  -0      1111
----    -------
```
No overflow

```
  -2      1101
+ -5    + 1010
----    -------
  -7     10111
----    +    1
        -------
          1000
        -------
```
No overflow

```
  -3      1100
+ -7    + 1000
----    -------
 -10     10100
----    +    1
        -------
          0101
        -------
```
Overflow!

在二进制加减法中，判断溢出是否发生取决于你是否在执行有符号的运算。溢出的概念主要适用于有符号数，通常是使用2's complement表示法。

对于加法：

1. **正溢出**：当你将两个正数相加并得到一个负结果时，发生正溢出。
2. **负溢出**：当你将两个负数相加并得到一个正结果时，发生负溢出。

对于减法： 由于减法可以被视为加法（减去一个数等同于加上它的负数），因此溢出条件与上述相同。

判断溢出的实际方法：

1. 检查操作数的符号和结果的符号。
2. 如果两个正操作数的加法得到一个负结果，或者两个负操作数的加法得到一个正结果，则发生溢出。

3. 更具体地说，可以检查进位到符号位和从符号位的进位。如果它们不同，就发生了溢出。例如，对于8位数，如果从第7位到第8位有进位，但从第8位向上没有进位（或相反），则发生溢出。

这种基于进位的方法在硬件加法器中更为实用，因为可以直接从加法器的输出中获得进位信号，用于溢出检测。

2.10.8 Excess Notation (Excess Representation)

- Besides sign-and-magnitude and complement schemes, the **excess representation** is another scheme
- It allows the range of values to be distributed **evenly** between the positive and negative values, by a simple translation (addition/subtraction)
- Example: Excess+4 (Excess-(-4)) representation on 3-bit numbers. See table below

Excess-4 Representation	*Value*
000	-4
001	-3
010	-2
011	-1
100	0
101	1
110	2
111	3

- Example: Excess+8 (Excess-(-7)) representation on 4-bit numbers

Excess-8 Representation	*Value*
0000	-8
0001	-7
0010	-6
0011	-5
0100	-4
0101	-3
0110	-2
0111	-1

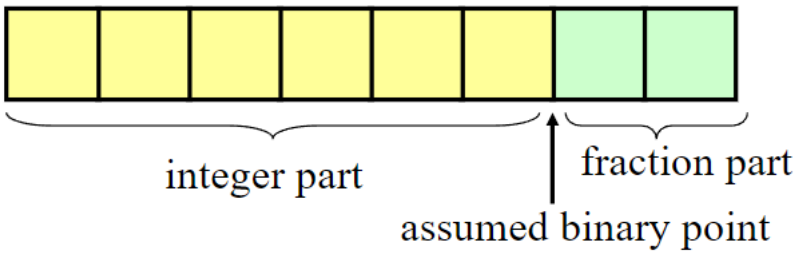
Excess-8 Representation	*Value*
1000	0
1001	1
1010	2
1011	3
1100	4
1101	5
1110	6
1111	7

2.11 Real Numbers

- Many applications involve computations not only on integers but also on real numbers
- How are real numbers represented in a computer system?
- Due to the finite number of bits, real number are often represented in their approximate values

2.11.1 Fixed-point Representation

- In **fixed-point representation**, the number of bits allocated for the whole number part and fractional part are fixed
- For example, given an 8-bit representation, 6 bits are for whole number part and 2 bits for fractional parts

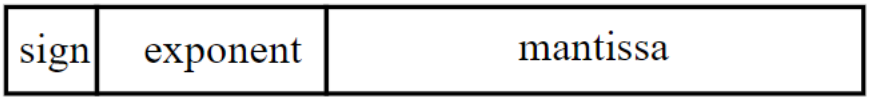


- If 2s-complement is used, we can represent values like:
  $011010.11_{2s} = 26.75_{10}$
  $111110.11_{2s} = -000001.01_2 = -1.25_{10}$

2.11.2 Floating-point Representation

- Floating-point representation has limited range
- Alternative: **Floating points numbers** allow us to represent very large or very small numbers
- Examples:
  - 0.23×10^{23} (very large positive number)
  - 0.5×10^{-37} (very small positive number)
  - -0.2397×10^{-18} (very small negative number)
- 3 components: **sign, exponent and mantissa (fraction)**
- The base (radix) is assumed to be 2
- Two formats:
  - Single-precision (32-bit): 1-bit sign, 8-bit exponent with bias 127 (excess-127), 23-bit mantissa
  - Double-precision (64-bit): 1-bit sign, 11-bit exponent with bias 1023 (excess-1023), 53-bit mantissa

3 components: **sign**, **exponent** and **mantissa (fraction)**

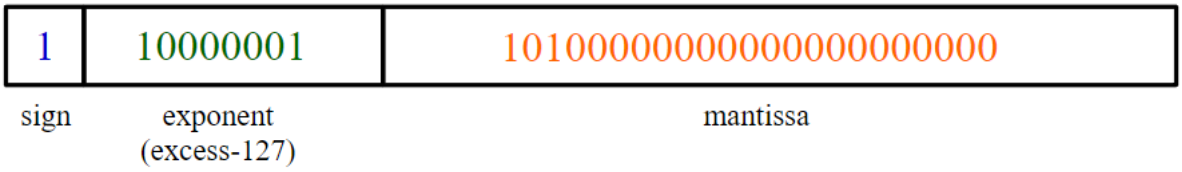


- Sign bit: 0 for positive, 1 for negative
- Mantissa is **normalized** with an implicit leading bit 1
  - 110.1_2 → normalized → $1.101_2 \times 2^2$ → only 101 is stored in the mantissa field
  - 0.00101101_2 → normalized → $1.01101_2 \times 2^{-3}$ → only 01101 is stored in the mantissa field

- Example: How is -6.5_{10} represented in IEEE 754 single-precision floating-point format?

$$-6.5_{10} = -110.1_2 = -1.101_2 \times 2^2$$

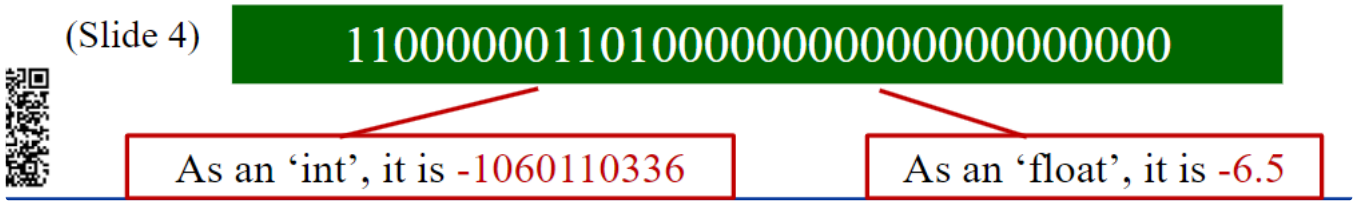
Exponent = 2 + 127 = 129 = 10000001_2



- We may write the 32-bit representation in hexadecimal:

$$1\ 10000001\ 10100000000000000000000_2 = C0D00000_{16}$$

(Slide 4)



3 - MIPS Assembly I

3.0 Recap

3.1 Instruction Set Architecture (ISA)

> 指令集架构（Instruction Set Architecture, ISA）定义了一个计算机系统可以执行的低级机器语言指令集，也就是计算机硬件能够理解和执行的指令。
>
> 指令集架构涵盖了以下几个方面：
>
> 1. **操作和指令**：定义了计算机能够执行的基本操作，如加法、减法、乘法、逻辑操作等。
> 2. **寄存器**：描述了计算机中的数据存储位置，通常分为通用寄存器、状态寄存器等。
> 3. **地址模式**：定义了如何计算数据和指令的存储位置。
> 4. **数据类型**：描述了支持的数据的种类和大小，如整数、浮点数等。
> 5. **异常和中断处理**：定义了当某些事件（如算术溢出、缺页中断）发生时计算机应该如何响应。
>
> 不同的指令集架构会导致计算机的性能、功耗、代码密度等方面的差异。有一些著名的ISA，如x86（由Intel和AMD使用）、ARM（用于大多数移动设备）、MIPS等。
>
> ISA是计算机架构的一个层次，通常分为三个层次：
>
> 1. **高级语言层**：如Python、Java等。
> 2. **汇编语言和指令集架构层**：这里就是ISA所在的层次。
> 3. **微架构或实现层**：这是具体硬件的实现细节，比如Intel的Core、Pentium等或AMD的Ryzen系列。
>
> ISA定义了软件与硬件之间的接口，使得软件开发者可以不必关心底层硬件的具体实现细节，而只需要关心指令集来编写程序。

- Instruction Set Architecture (ISA) (指令集架构)
  - An abstraction on the interface between the hardware and the low-level software
    - Software: To be translated to the instruction set
    - Hardware: Implementing the instruction set
  - ISA Allows computer designers to talk about functions independently from the hardware that performs them
    - > 允许计算机设计师在不考虑特定硬件实现的情况下，讨论和设计计算机功能。以指令集架构为例，设计师可以定义一个指令来完成特定的操作，例如加法，而不需要指定这个加法是如何在硬件上实现的。这样，ISA就充当了软件和硬件之间的桥梁，为高级编程语言提供了一个稳定的接口。
  - This abstraction allows many implementations of varying cost and performance to run identical software
    - > 由于存在上述的抽象，同一个指令集架构可以有多种不同的硬件实现。这些实现可能在成本和性能上有所不同。例如，高性能的服务器处理器和低功耗的移动设备处理器可能都遵循相同的ISA，但它们在微架构（即具体的硬件实现）上会有所不同。尽管如此，由于它们共享相同的ISA，它们仍然可以运行相同的软件。这种抽象确保了软件的兼容性和长期稳定性。

3.2 Machine Code vs. Assembly Language

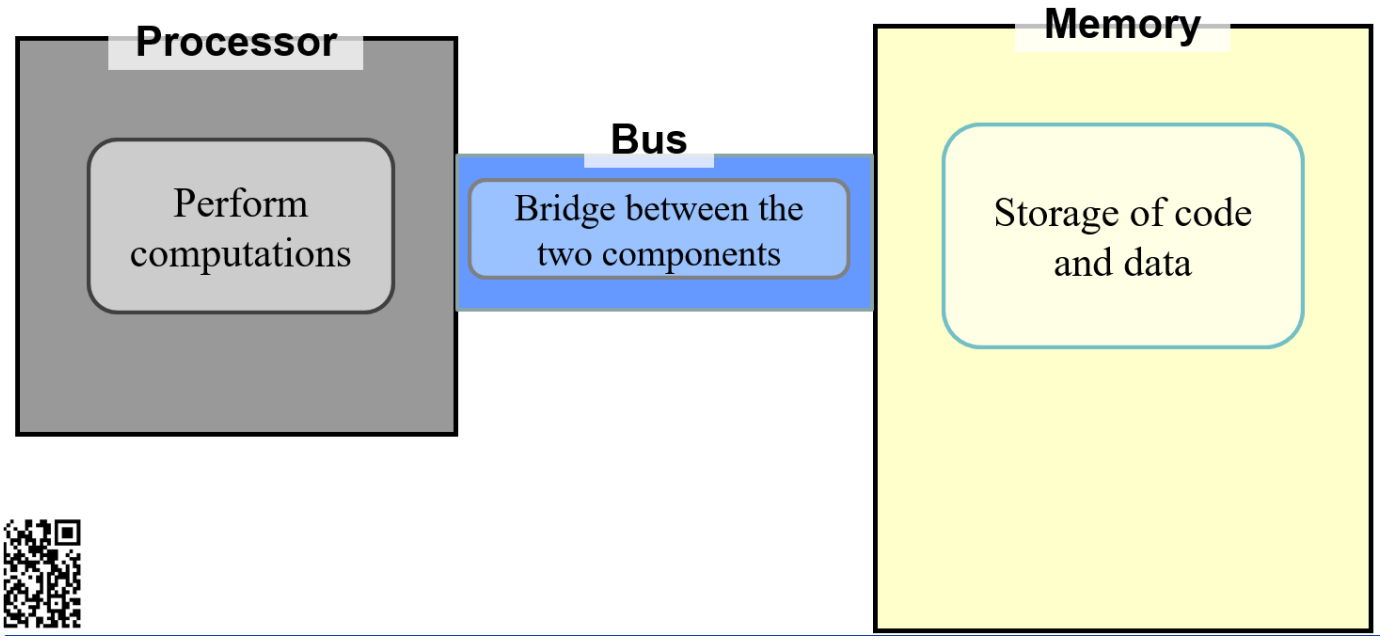
- Machine Code
  - Instructions are represented in **Binary**
  - Hard and tedious for programmer
- Assembly Language
  - Symbolic version of machine code
  - Human readable
    - **1000110010100000** in binary and `add A,B` in assembly language
  - **Assembler** translates from assembly language to machine code
  - Assembly can provide '**pseudo-instructions**' as **syntactic sugar**
    - "伪指令" (pseudo-instructions)
      - 在汇编语言中，伪指令不是实际的机器语言指令，但它们在汇编器中有特定的含义。汇编器在处理伪指令时会将它们转换为一个或多个实际的机器指令，或执行特定的操作。例如，某些伪指令可能用于数据分配或指定一个内存地址。

      "语法糖" (syntactic sugar)
      - 语法糖是指在编程语言中，为了使代码更易读、更易写而添加的某种语法。这种语法并没有为语言增加新的功能，但它为编程师提供了一种更加方便、更加直观的方式来表示某个操作或结构。

      汇编语言提供的伪指令可以被视为一种语法糖，因为它们为程序员提供了一种更简单、更直观的方式来编写汇编代码，尽管这些伪指令在最终转换为机器代码时可能会被替换为实际的指令或执行一系列操作。简言之，伪指令使得汇编代码更易读和更易写，但它们并不直接对应于实际的硬件指令。
  - When considering performance, only read instructions are counted, pseudo-instructions are not counted

3.3 Walkthrough

The components

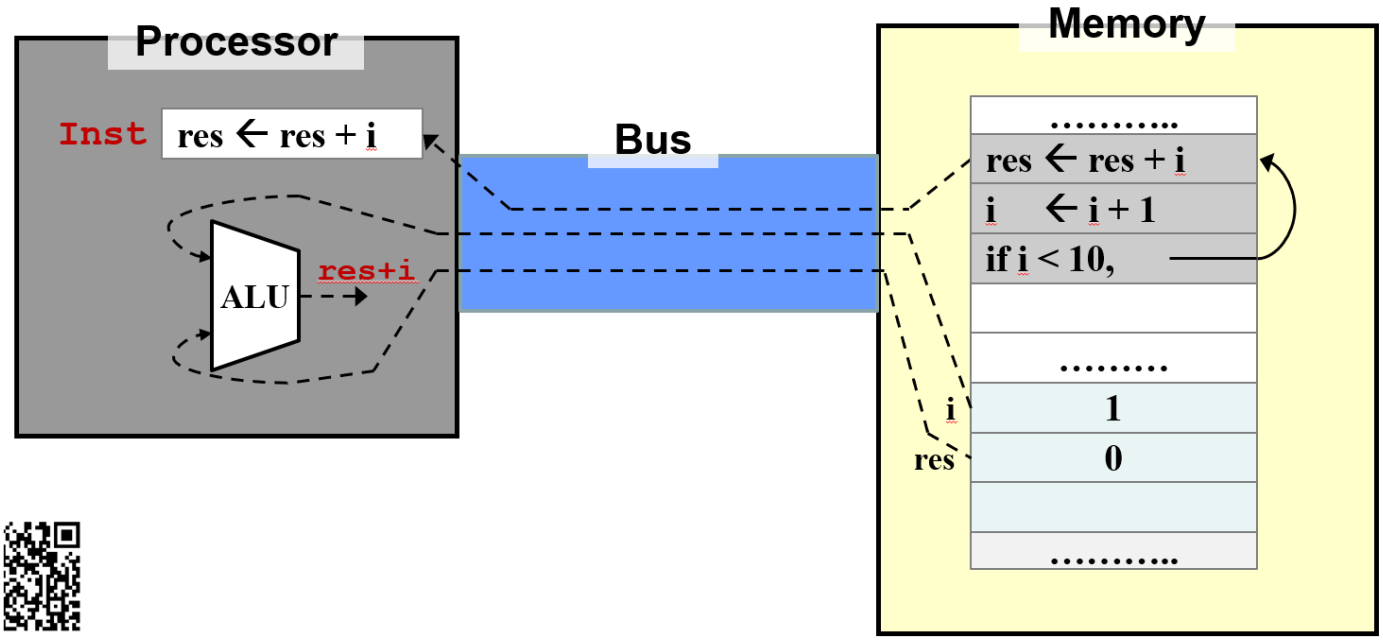


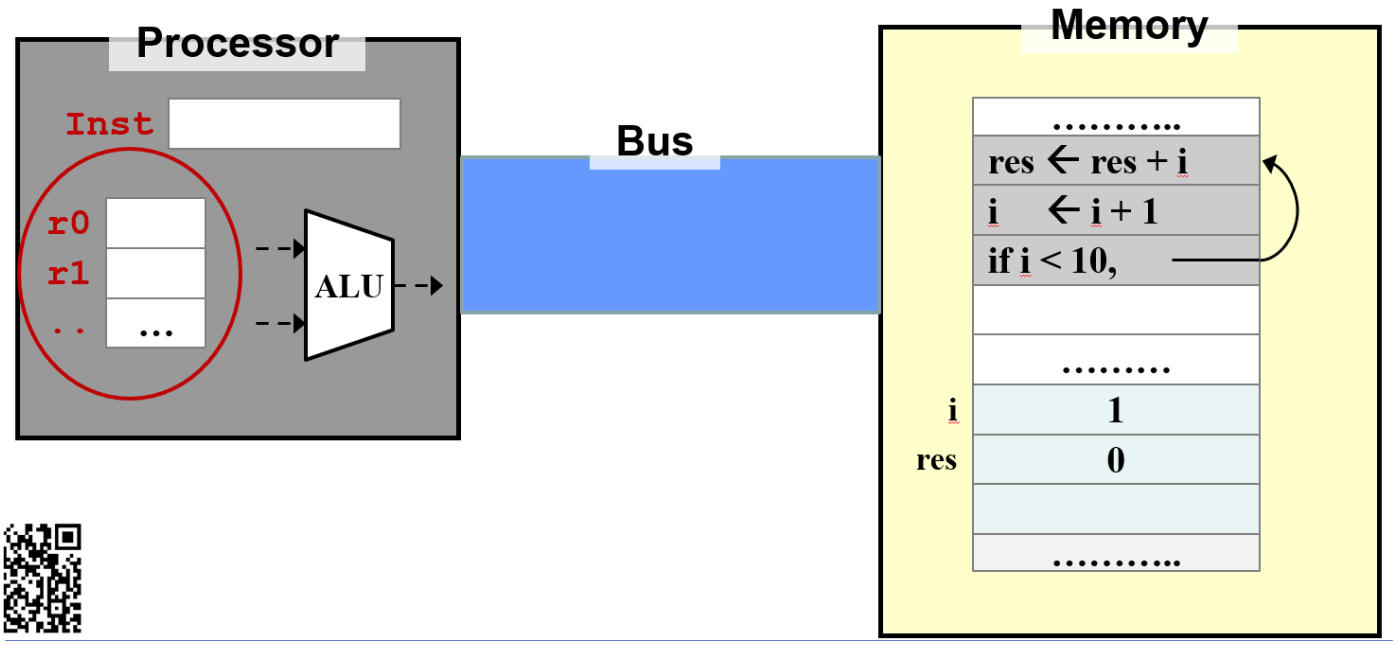
- Assume a simple computing-storage platform, including a processor, bus and a memory
  - Processor processing instructions
  - Bus transmit the data between memory and processor
  - Memory store the instructions and temp data

The code in Action

```
for (i=1; i<10; i++) {
    res = res + i;
}
```

```
res ← res + i
i ← i + 1
if i < 10, repeat
```

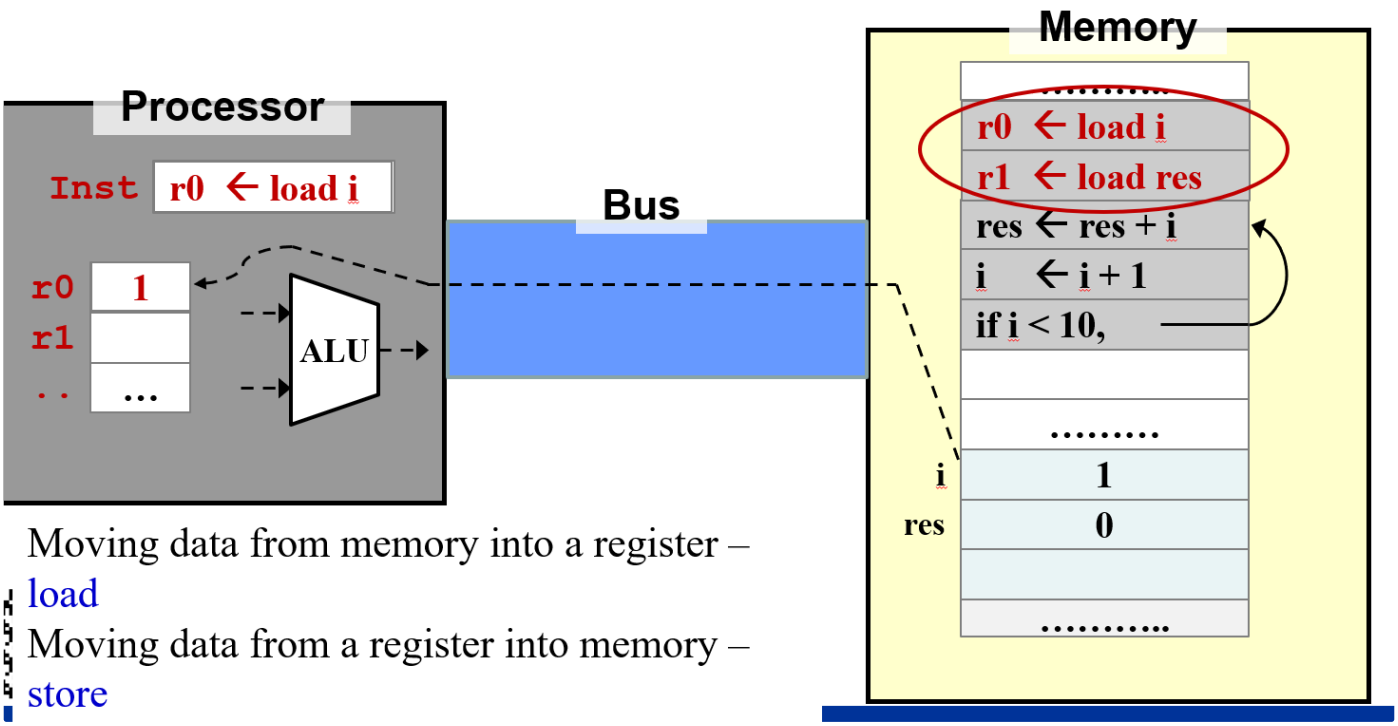




- The instructions and variables are stored in the **memory**
- The variable, `i` and `res` has been transmitted to **processor** through **bus**
  - The data of two variables and the instruction has been stored in **CPU register**, in order to get faster speed
  - > - The register (寄存器) is inside in the CPU, which is using to store the processing data and instruction
    > - The register can provide data and instruction quickly to the ALU
    > - The size of register determines the bit of the CPU, like 32 bit or 64 bit CPU
    > - THE REGISTER IS NOT L1, L2, and L3 CAHCE
    > - L1, L2 and L3 cache is bigger, but slower than register. It provides a buffer between the memory and the processor register, mainly to decrease the distance between these two, from the physically abstraction.

Memory Instructions

- **Need instruction to move data into registers**
  - Also to move data from registers to memory later

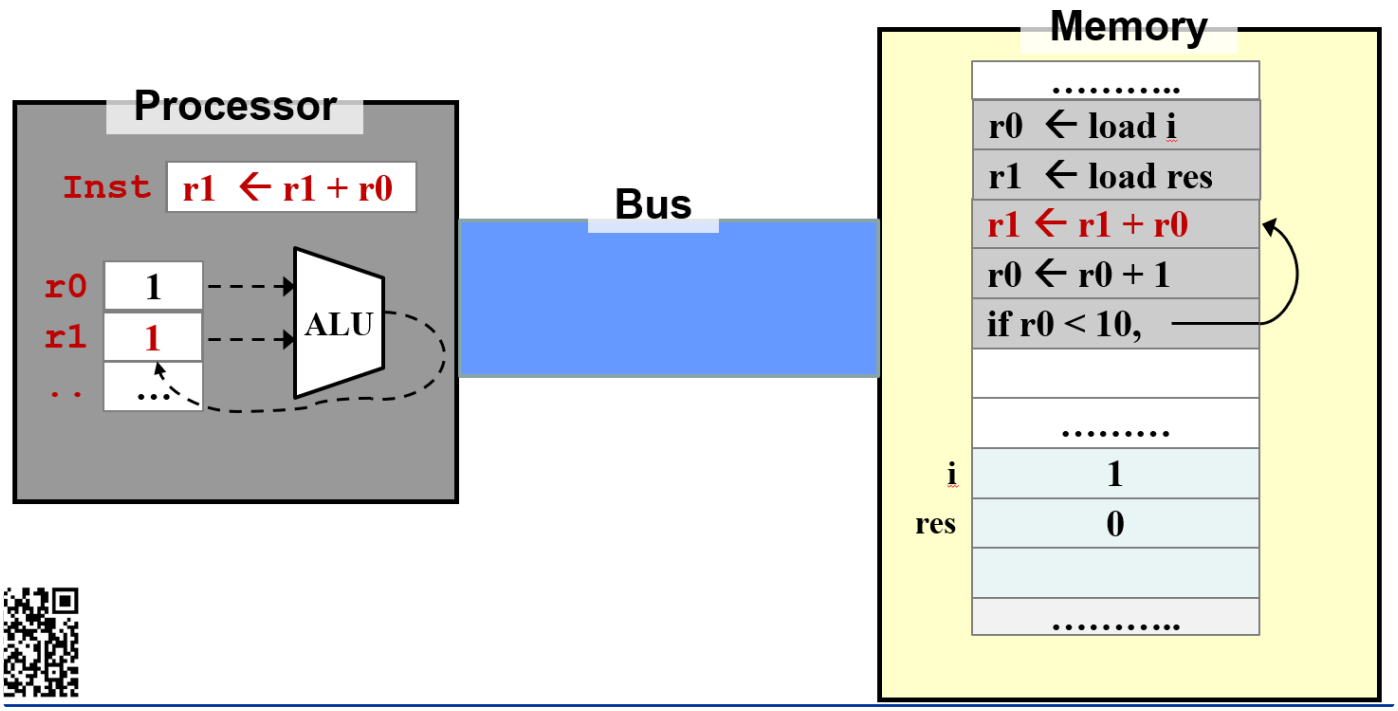


Moving data from memory into a register – load

Moving data from a register into memory – store

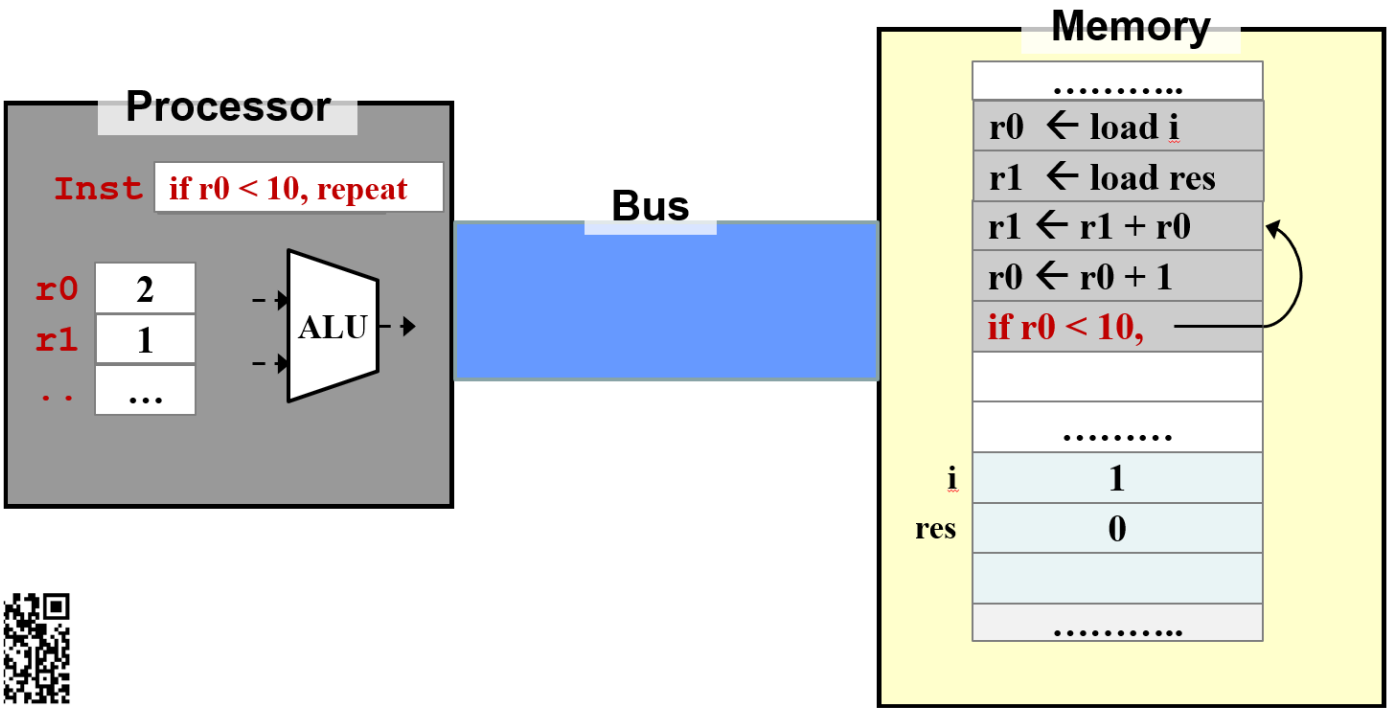
- The machine needs instruction to move data from memory to the register
  - `r0 ← load i` is in the memory
  - This instruction has been transmitted to the processor through bus
  - Then the register `0` is **loaded** the data of variable `i`
- Then the same process for variable `res`

Reg-to-Reg Arithmetic



- After moving all needed variables into the register, the ALU can load all need variables from the register, but not memory. Which is much faster
- Sometimes the arithmetic operation uses a constant value instead of register value
  - `r0 ← r0 + 1`, `1` is the constant value
  - > 常数被称为"立即数"（Immediate Value）。立即数是直接编码在机器指令中的常数值。
    >
    > 所以，当ALU要执行加法操作时，它不需要从寄存器或内存中加载立即数，因为这个值已经直接包含在指令中了。这意味着CPU可以在执行指令的同时，直接从指令本身获取这个常数值，并在ALU中与寄存器中的值进行计算。

Execution Sequence (Loop)

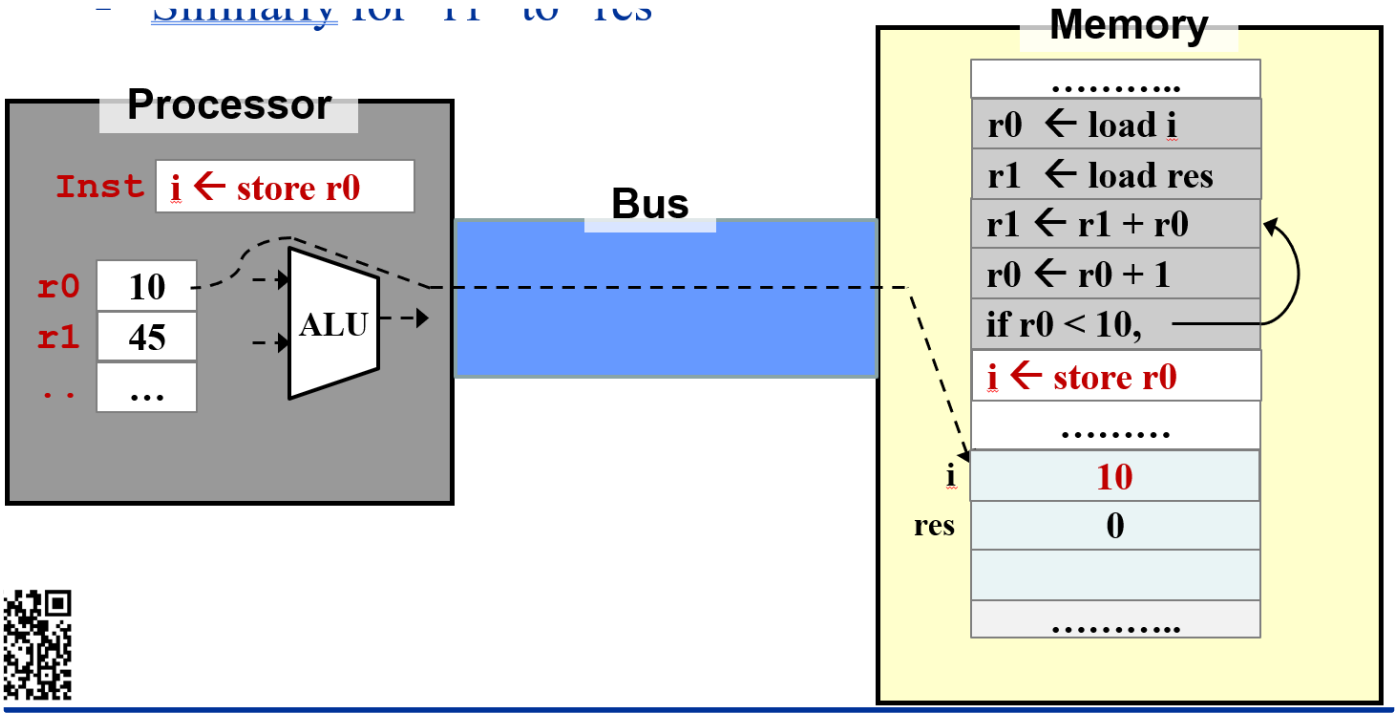


- What happened when the machine want to process the loop?

```
1 res ← res + i
2 i ← i + 1
3 if i < 10, repeat
```

- We need instructions to change the **control flow** based on condition:
  - Repetition (loop) and Selection (if-else) can both support
- Since the condition succeeded, execution will repeat from the indicated position
  - The execution will continue sequentially
  - Until we see another control flow instruction

Memory Instructions (Finish Computing, output the result)



- Finally, move the computed value out from the register, store it into the memory

Summary

- The stored-memory concept:
  - Both **instruction** and **data** are stored in memory
- **The load-store model:**
  - Limit memory operations and relies on registers for storage during execution
- **The major types of assembly instruction:**
  - **Memory:** Move values between memory and registers
  - **Calculation:** Arithmetic and other operations
  - **Control flow:** Change the sequential execution

3.4 General Purpose Registers

General Purpose Registers（GPR，通用寄存器）是中央处理器（CPU）内部的一组寄存器，这些寄存器不是为某个特定的任务或操作而设计，而是为了存储临时数据或在指令执行过程中用作操作数。通常，汇编语言编程或机器语言编程中的指令可以直接访问和操作这些寄存器。

GPR的特点和用途

1. **多功能**：与专用寄存器（如浮点寄存器或状态寄存器）相比，通用寄存器可以用于多种任务，如算术运算、数据移动、逻辑操作等。
2. **速度**：访问通用寄存器的速度非常快，因为它们是CPU内部的存储单元。这使得它们成为临时存储操作数和结果的理想选择。
3. **数量**：现代处理器通常具有多个GPR。例如，x86架构提供了EAX、EBX、ECX、EDX等寄存器（在64位模式下，这些寄存器分别被称为RAX、RBX、RCX、RDX）；而ARM架构提供了R0到R15的寄存器。
4. **扩展性**：随着处理器架构的发展，GPR的数量和大小可能会发生变化。例如，早期的x86处理器使用16位的AX、BX、CX和DX作为其GPR，而现代的x86-64处理器则使用64位的RAX、RBX、RCX和RDX。

5. **任务**：尽管这些寄存器被称为"通用"，但在某些指令或情境下，它们可能有特定的用途或约定。例如，在某些架构中，某些GPR可能首选或专用于函数调用的返回值或作为参数传递。

- Not all CPU register are GPR
  - CPU寄存器是一个广泛的类别，涵盖了处理器内的所有寄存器。这包括GPR，但也包括其他专用或特定功能的寄存器。
  - GPR是处理器中的寄存器，可以用于多种通用计算和数据传输任务。它们并没有为特定功能（如浮点运算）专门设计。
- Data are transferred from memory to registers for faster processing
- A typical architecture has 16 to 32 registers
- GPR has no data type

- There are **32 registers** in MIPS assembly language
  - Can be referred by a number ($0, $1,…, $31) OR
  - referred by name ($a0, $t1)

Name	Register number	Usage
$zero	0	Constant value 0
$v0-$v1	2-3	Values for results and expression evaluation
$a0-$a3	4-7	Arguments
$t0-$t7	8-15	Temporaries
$s0-$s7	16-23	Program variables

Name	Register number	Usage
$t8-$t9	24-25	More temporaries
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address

- $at (register 1) is reserved for the assembler
- $k0-$k1 (register 26-27) are reserved for the operation system

3.5 MIPS Assembly Language

MIPS assembly language 是 MIPS 架构的汇编语言。MIPS (Microprocessor without Interlocked Pipeline Stages) 是一种基于RISC (Reduced Instruction Set Computer) 原则的微处理器架构，旨在简化指令集以提高性能。以下是有关MIPS汇编语言的一些关键点：

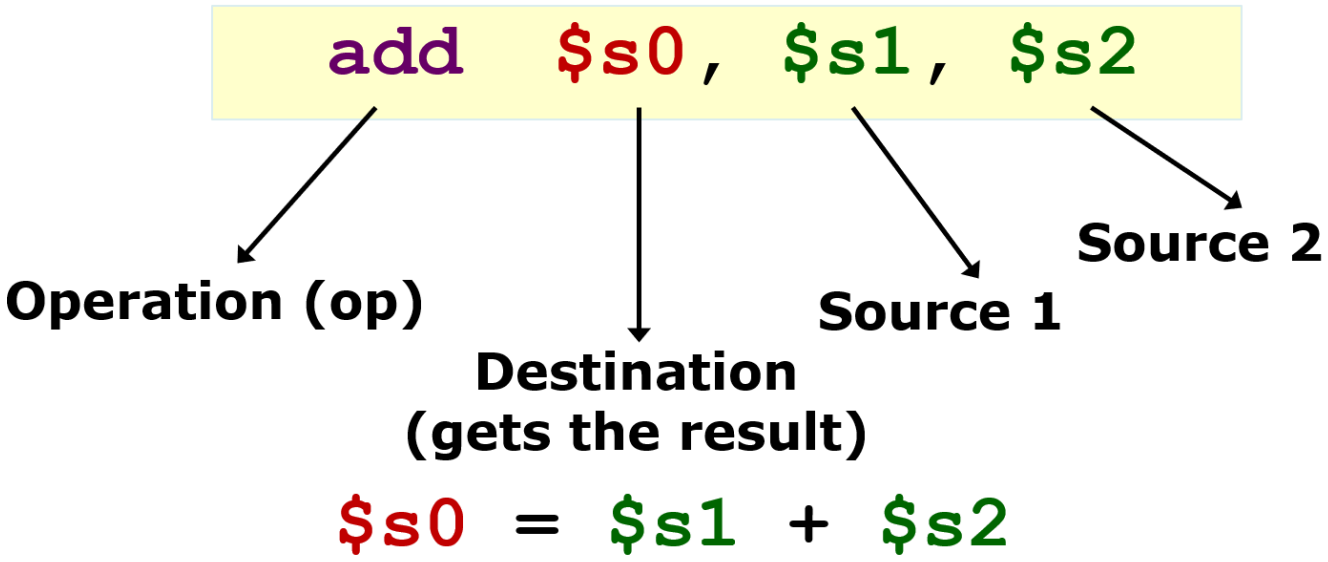
1. **指令集**：MIPS 指令集被设计得相对简单，并与其硬件管道设计紧密相连。这使得MIPS能高效地执行指令，同时简化了硬件实现。
2. **寄存器**：MIPS架构具有32个通用寄存器，标记为 $0 到 $31 。其中， $0 寄存器总是存储值0，其他寄存器用于不同的目的。例如， $sp 是堆栈指针， $ra 是返回地址寄存器。
3. **指令格式**：MIPS指令有几种格式，最常见的是R型、I型和J型。不同的格式支持不同的操作，例如算术运算、数据传输和跳转。

4. **指令示例**：以下是一些常见的MIPS汇编指令示例：
    - `add $t0, $t1, $t2`：将 `$t1` 和 `$t2` 中的值加在一起，并将结果存储在 `$t0` 中。
    - `lw $t0, 4($sp)`：从堆栈指针 `$sp` 加上偏移量 4 的位置加载一个字（word）到 `$t0`。
    - `beq $t0, $t1, label`：如果 `$t0` 和 `$t1` 的值相等，则跳转到 `label`。

- Each instruction executes a simple command
    - Usually has a counterpart in high level programming language like C/C++, Java
- Each line of assembly code contains at most 1 instruction
- # (hex-sign) is used for comments
    - Anything from # to end of line is a comment and will be ignored by the assembler

```
add $t0, $s1, $s2   # $t0 ← $s1 + $s2
sub $s0, $t0, $s3   # $s0 ← $t0 - $s3
```

General Instruction Syntax



- Naturally, most of the MIPS arithmetic/logic operations have three operands: 2 sources and 1 destination

Arithmetic Operation: Addition

C Statement	MIPS Assembly Code
`a = b + c;`	`add $s0, $s1, $s2`

- The value `a`, `b` and `c` are loaded into the CPU register `\$s0`, `\$s1` and `\$s2`
    - This procession is called **variable mapping**

> MIPS arithmetic operations are mainly **reg-to-reg**

Arithmetic Operation: Subtraction

C Statement	MIPS Assembly Code

`a = b - c;`	`sub $s0, $s1, $s2`
	`$s0` → variable `a`
	`$s1` → variable `b`
	`$s2` → variable `c`

- Similar variable mapping processing than addition
  - The position of `\$s1` and `\$s2` is important for subtraction

Complex Expression

C Statement	MIPS Assembly Code

`a = b + c - d;`	`??? ??? ???`
	`$s0` → variable `a`
	`$s1` → variable `b`
	`$s2` → variable `c`
	`$s3` → variable `d`

- A single MIPS instruction can handle at most two source operands

➔ **Need to break a complex statement into multiple MIPS instructions**

MIPS Assembly Code
`add $t0, $s1, $s2  # tmp = b + c` `sub $s0, $t0, $s3  # a = tmp - d`

Use temporary registers **$t0** to **$t7** for intermediate results

- A single MIPS instruction can only handle at most two source operands.
  - In this case, we should break this expression into two separate expressions, one is addition and another is subtraction.

C Statement	Variable Mappings

`f = (g + h) - (i + j);`	`$s0` → variable `f`
	`$s1` → variable `g`
	`$s2` → variable `h`
	`$s3` → variable `i`
	`$s4` → variable `j`

- Break it up into multiple instructions
  - Use two temporary registers **`$t0`**, **`$t1`**

```
add $t0, $s1, $s2   # tmp0 = g + h
add $t1, $s3, $s4   # tmp1 = i + j
sub $s0, $t0, $t1   # f = tmp0 - tmp1
```

Constant/Immediate Operands

C Statement	MIPS Assembly Code
`a = a + 4;`	`addi $s0, $s0, 4`

- Immediate value are also called: "constant value"
- The instruction is "Add immediate" (`addi`), differ from the default addition `add`

Register zero ($0)

C Statement	MIPS Assembly Code

`f = g;`	`add $s0, $s1, $zero`
	
	`$s0` → variable `f`
	`$s1` → variable `g`

- The number zero (0) is **constantly** assigned at register zero (`\$0` or `\$zero`)
- The above assignment is equivalent to the pseudo instruction (move)
  - `add \$s0, \$s1, \$zero`
  - `move \$s0, \$s1`

3.6 Logical Operations

Overview

- Arithmetic instructions view the content of a register as a single quantity (singed or unsigned integer)
- The logical operations allows the ALU to view register as 32 raw bits rather than a single 32-bit number

Logical operation	C operator	Java operator	MIPS instruction
Shift Left	<<	<<	sll
Shift right	>>	>>, >>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR	|	|	or, ori
Bitwise NOT*	~	~	nor
Bitwise XOR	^	^	xor, xori

- Truth table of logical operations

AND

a	b	a AND b
0	0	0
0	1	0
1	0	0
1	1	1

OR

a	b	a OR b
0	0	0
0	1	1
1	0	1
1	1	1

NOR

a	b	a NOR b
0	0	1
0	1	0
1	0	0
1	1	0

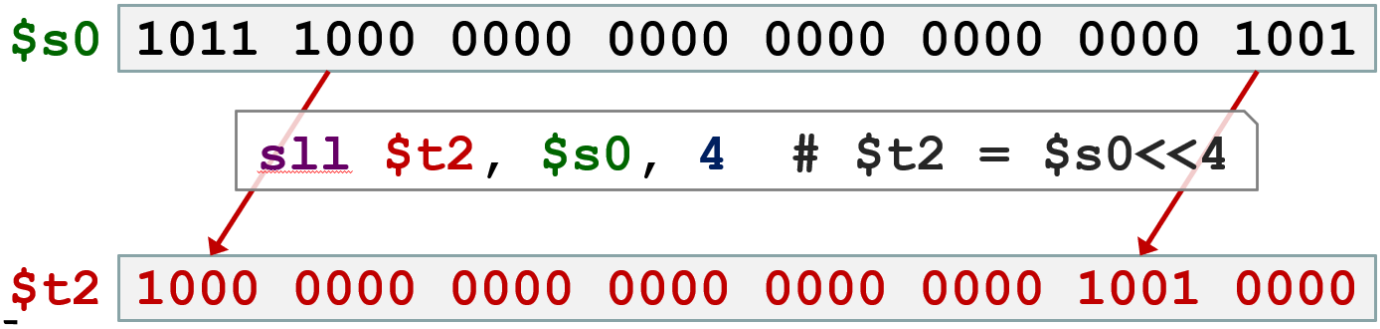
XOR

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

Shifting

Opcode: `srl` (shift right logical)

Shifts right and fills emptied positions with zeroes.



- Left shift: `sll`
- The above code let the value in register `\$s0` **left shift** 4 bits, then store into the register `\$t2`

C Statement	MIPS Assembly Code
`a = a * 8;`	`sll $s0, $s0, 3`

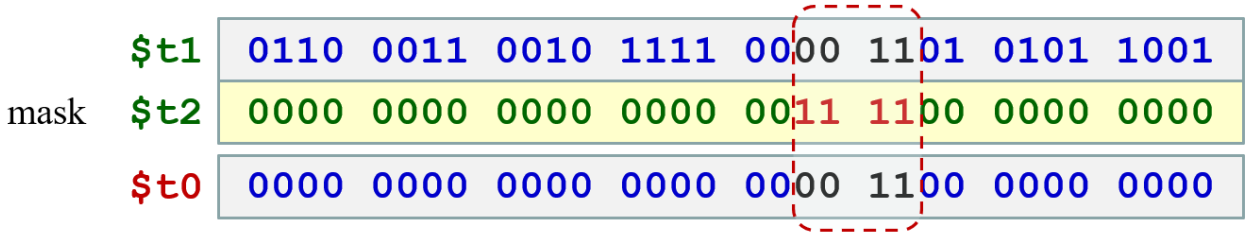
- Right shift: `srl`

Bitwise AND

Opcode: `and` (bitwise AND)

Bitwise operation that leaves a 1 only if both the bits of the operands are 1

- E.g.: `and $t0, $t1, $t2`



- Bitwise AND: `and`
  - Do AND operation on the register `\$t1` and `\$t2` **bit by bit**, then store the result to register `\$t0`
  - It can be used for *masking operation*

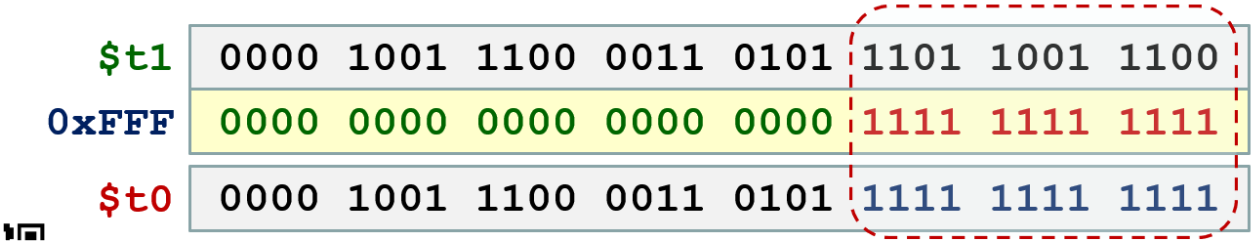
Bitwise OR

Opcode: `or` (bitwise OR)

Bitwise operation that that places a 1 in the result if either operand bit is 1

Example: `or $t0, $t1, $t2`

- The `or` instruction has an immediate version `ori`
- Can be used to force certain bits to 1s
- E.g.: `ori $t0, $t1, 0xFFF`



- Bitwise OR: `or`
- Similar usage with the Bitwise AND operator
  - Also can be used to *mask*

Bitwise NOR

- Strange fact 1:
  - There is no **NOT** instruction in MIPS to toggle the bits (1 → 0, 0 → 1)
  - However, a **NOR** instruction is provided:

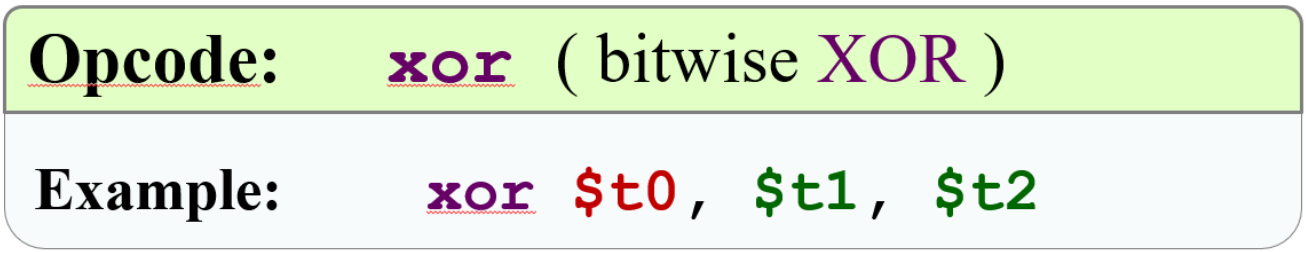
Opcode: `nor` (bitwise NOR)

Example: `nor $t0, $t1, $t2`

- Bitwise NOR: `nor`
- Similar usage with the Bitwise AND, OR operator
- There is no NOT instructions in MIPS to toggle bits 1→0, 0→1
  - However, the `nor` instruction can achieve the NOT operation
    - `nor \$t0, \$t0, \$zero`
    - This instruction turn all 0 to 1, and all 1 to 0 in register `\$t0`

- There DO NOT exist `nori` instruction

Bitwise XOR



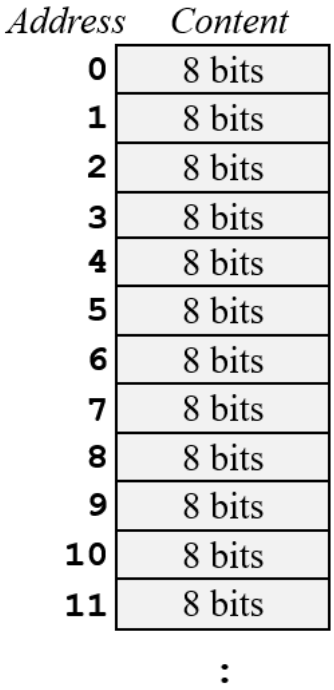
- Bitwise XOR: `xor`
- To get `not` operation through `xor`:
  - `xor \$t0, \$t0, \$t2`
- There DO exist `xori`

3.7 Large Constant: Case Study

4 - MIPS II

4.1 Memory Organization

- The main memory can be viewed as a large, single-dimension array of memory locations
- Each location of the memory has an address, which is an index into the array
  - Given a `k`-bit address, the address space is of size 2^k
- The memory map on the below contains one byte (8 bits) in every location/address.
  - This is called **byte addressing**



Byte Addressing 字节寻址
Byte addressing（字节寻址）是指在计算机内存中，每个字节都有其独特的地址。这意味着，即使一个数据项（例如32位整数）需要多个字节来存储，我们仍然可以分别访问这些字节。

为了更加清晰地说明，我们可以拿一个32位整数来做例子：

假设我们有一个32位的整数，它需要4个字节来存储（因为32位等于4字节）。在byte addressing系统中，这4个字节会被存放在连续的内存地址中。假设该整数的第一个字节存储在地址0x1000处，那么：

- 第一个字节的地址是：0x1000
- 第二个字节的地址是：0x1001
- 第三个字节的地址是：0x1002
- 第四个字节的地址是：0x1003

这就是byte addressing的核心思想，即每个字节在内存中都有唯一的地址。这种方式使得硬件和软件可以非常灵活地访问内存，但与此同时，也需要在内存管理方面投入更多的精力，以确保有效地使用这种精细级别的寻址机制。

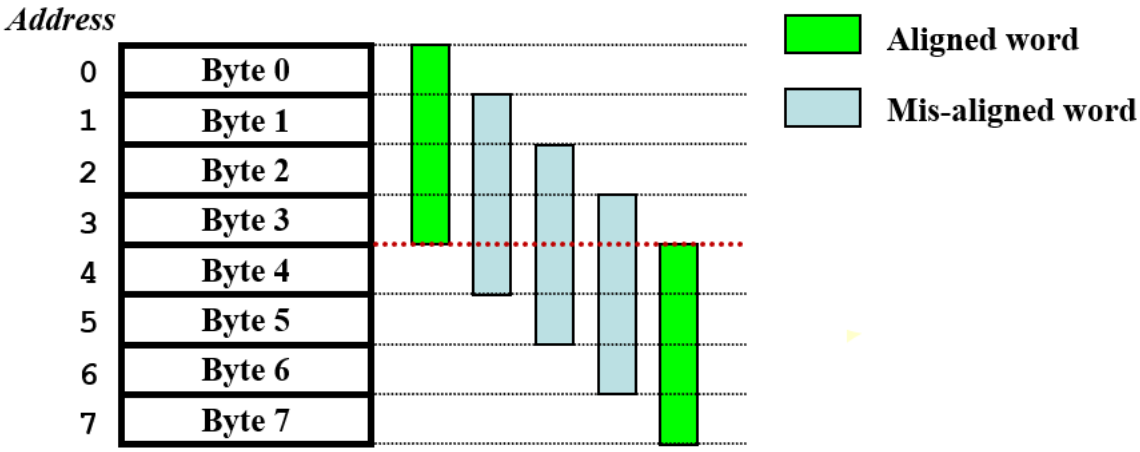
4.1.1 Memory: Transfer Unit

- Using distinct memory address, we can access:
  - a single byte (byte addressable) or
  - a single word (word addressable)
- Word is:
  - Usually 2^n bytes
  - The common unit of transfer between processor and memory
  - Also commonly coincide with the register size, the integer size and instruction size in most architecture

> Word的大小通常与寄存器大小相同，例如32位架构中，通常由32位的寄存器（4字节），一个word就是32位

4.1.2 Memory: Word Alignment

- **Word alignment**:
  - Words are aligned in memory if they begin at a byte address that is a multiple of the number of bytes in the word
- Example: If a word consists of 4 bytes, then:



> "Word Alignment"（或简称 "Alignment"）是计算机存储和内存管理中的一个概念，它指的是数据项在内存中的开始地址应该是其大小（通常是数据项大小或特定架构的word大小）的某个倍数。
>
> 为什么需要对齐?
>
> 1. **性能**：在许多架构上，访问对齐的数据比非对齐的数据要快。当数据对齐时，数据可能完全位于一个或多个缓存行内，从而减少了需要访问的缓存行数量。
> 2. **硬件要求**：一些处理器不支持非对齐的数据访问，或者在尝试这样做时可能导致性能损失或异常。
>
> 以32位系统为例，其中word的大小为4字节（32位）：
>
> - 如果一个32位的整数地址为0x1004或0x1008，那么这个整数是对齐的，因为这些地址都是4的倍数。
> - 但是，如果这个32位的整数的地址为0x1005或0x1006，那么它就不是对齐的，因为这些地址不是4的倍数。
>
> 为了确保对齐，编译器和内存分配器通常会自动处理数据对齐的问题，为变量分配适当对齐的地址。但在低级编程或嵌入式系统开发中，程序员可能需要更加关注对齐的问题，因为它可能会影响性能或正确性。

如果是64位系统，则应该是8的倍数。

4.2 MIPS Memory Instructions

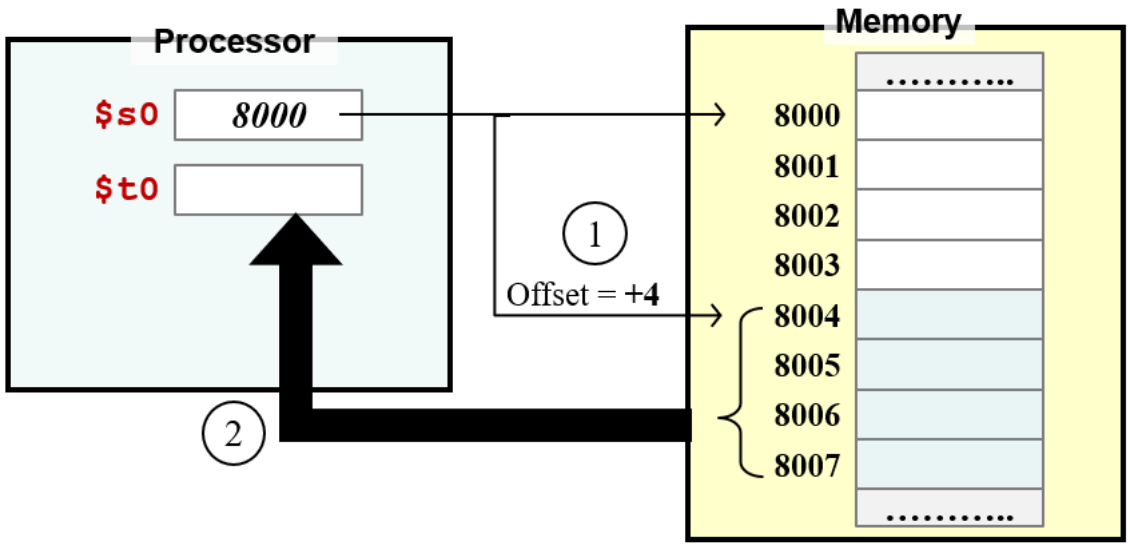
- MIPS is a load-store register architecture
  - 32 registers, each 32-bit (4 bytes) long
  - Each word contains 32-bit (4 bytes)
  - Memory addresses are 32-bit long

Name	Examples	Comments
32 registers	$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at	Fast processor storage for data. In MIPS, data must be in registers to perform arithmetic.
2^{30} memory words	Mem[0], Mem[1], …, Mem[4294967292]	Accessed only by data transfer instructions. MIPS uses **byte addresses**, so consecutive words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls

- 32 registers:
  1. $0 或 $zero：这个寄存器始终包含值0，任何尝试向其写入的操作都会被忽略。
  2. $1 或 $at：为汇编器预留的临时寄存器。
  3. $2-$3 或 $v0-$v1：用于返回函数值的寄存器。
  4. $4-$7 或 $a0-$a3：用于传递函数参数的寄存器。
  5. $8-$15、$24-$25 或 $t0-$t7 和 $t8-$t9：临时寄存器，函数调用不会保存它们。
  6. $16-$23 或 $s0-$s7：保存的寄存器，函数调用会保存它们。
  7. $26-$27 或 $k0-$k1：为操作系统预留的寄存器。
  8. $28 或 $gp：全局指针。
  9. $29 或 $sp：堆栈指针。
  10. $30 或 $fp：帧指针（在某些约定中使用）。
  11. $31 或 $ra：返回地址。
- memory words
  - 指内存中的数据单元。一个 "word" 在 MIPS 中通常指代一个固定大小的数据块，其大小在传统的 MIPS 架构中为 32 位，或 4 字节。
  - 如果 MIPS 系统有 2^{30} 个 "memory words"，这意味着这个系统有 2^{30} 个独立的 32 位数据块。换句话说，该系统的总内存大小是 2^{30}×32 位，或 2^{30}×4 字节（4GB）。
- Words and Memory words
  - **Word**：是指数据的大小。在 32 位 MIPS 架构中，一个 word 是 32 位或 4 字节。
  - **Memory Words**：是指内存中的 word 单元的数量。如果一个系统有 2^{30} 个 memory words，那么它具有 2^{30} 个独立的 32 位数据单元。

4.2.1 Memory Instruction: Load Word

- `lw $t0, 4($t0)`



- Steps:
  1. Memory Address = `$s0 + 4 = 8000 + 4 = 8004`
  2. Memory word at `Mem[8004]` is loaded into `$t0`

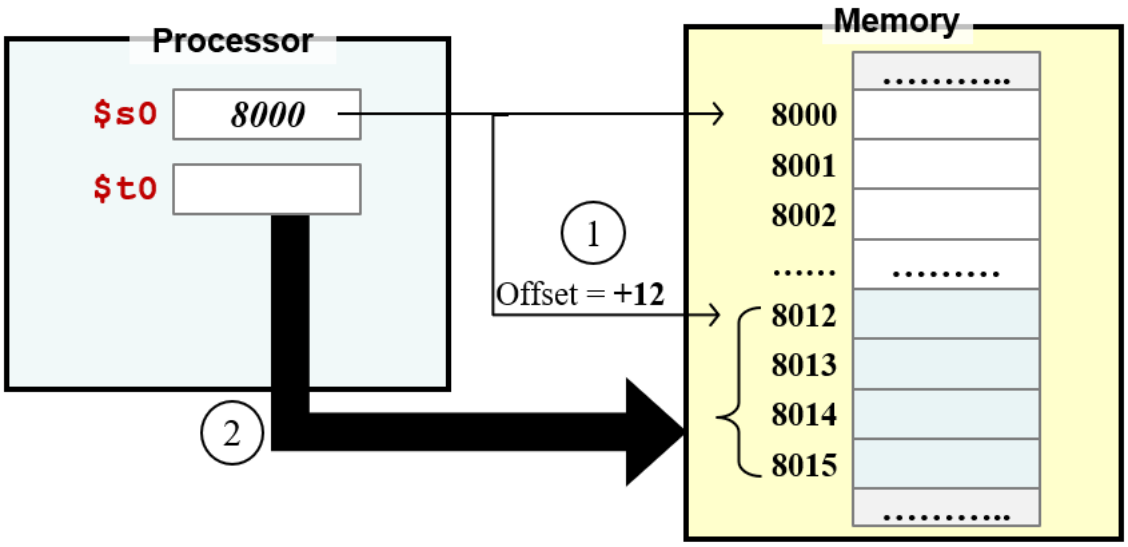
1. `lw`：这是 "load word" 的指令，意味着从内存中加载一个 32 位的数据块。
2. `$t0`：这是目标寄存器，指示数据从内存加载到哪个寄存器中。
3. `4($t0)`：这是源操作数，表示内存的地址。这个地址是通过取 `$t0` 寄存器中的值，并加上偏移量 `4` 来计算的。这里的偏移量是以字节为单位的，因此 `4` 实际上是 4 字节的偏移量。

所以，整体上，这条指令的意思是：从地址为 `$t0 + 4` 的位置加载一个 word（32 位的数据块）到 `$t0` 寄存器中。

例如，假设 `$t0` 中原来的值是 `0x8000`，那么这条指令会从内存地址 `0x8004` 加载一个 word 到 `$t0` 寄存器中。

4.2.2 Memory Instruction: Store Word

- `sw $t0, 12($s0)`



- Steps:
  1. Memory Address = `$t0 + 12 = 8000 + 12 = 8012`
  2. Content of `$t0` is stored into word at `Mem[8012]`

1. `sw`：这是 "store word" 的指令，意味着将一个 32 位的数据块存储到内存中。
2. `$t0`：这是源寄存器，它表示要存储到内存中的数据来源于哪个寄存器。
3. `12($s0)`：这是目标操作数，表示内存的地址。这个地址是通过取 `$s0` 寄存器中的值，并加上偏移量 `12` 来计算的。这里的偏移量是以字节为单位的，所以 `12` 实际上是 12 字节的偏移量。

因此，整体上，这条指令的意思是：将 `$t0` 寄存器中的 word（32 位的数据块）存储到地址为 `$s0 + 12` 的内存位置。

例如，假设 `$s0` 中的值是 `0x8000`，那么这条指令会将 `$t0` 寄存器中的内容存储到内存地址 `0x8012` 的位置。

4.2.3 Load and Store Instructions

- Only `load` and `store` instructions can access data in memory
- Example: Each array element occupies a word

C Code	MIPS Code

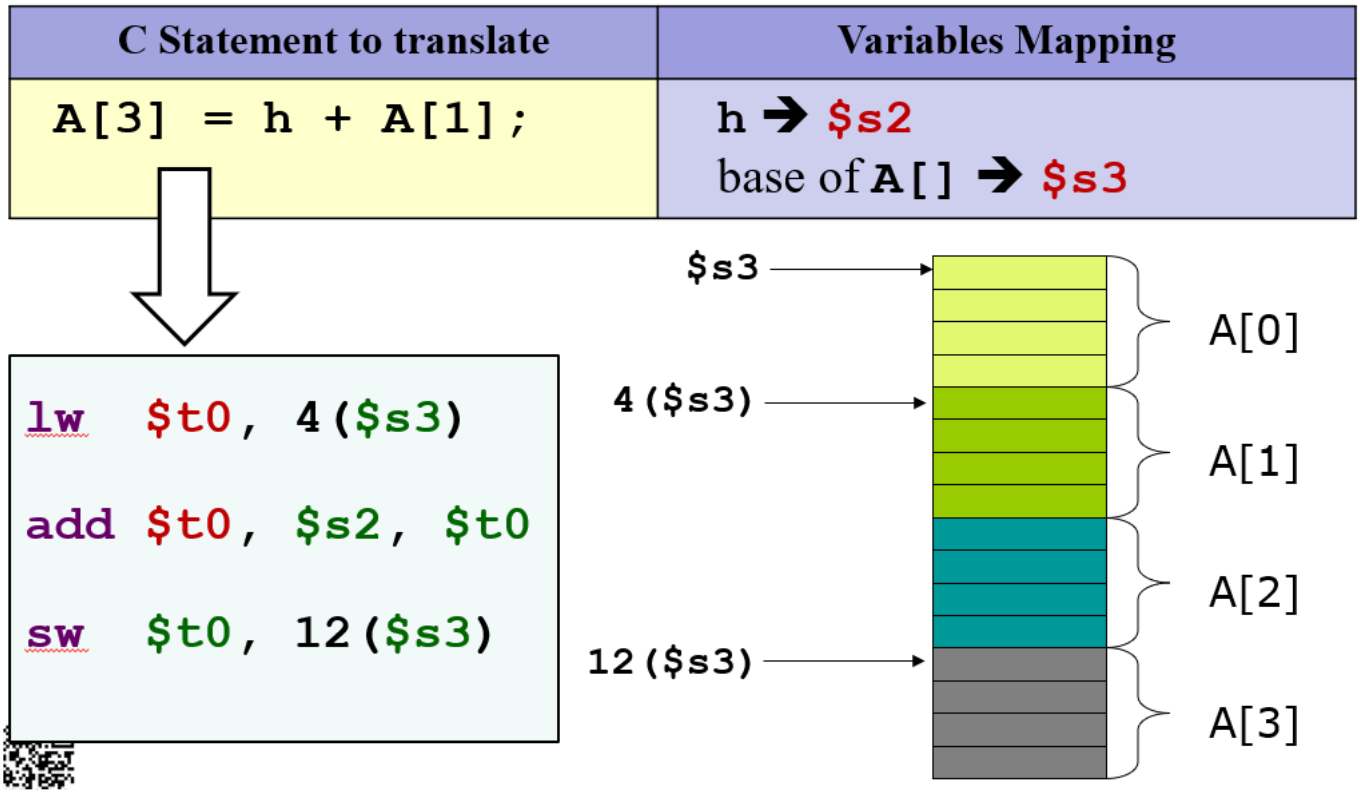
`A[7] = h + A[10]`	`lw $t0, 40($s3)`
	`add $t0, $s2, $t0`
	`sw $t0, 28($s3)`

- Each array element occupies a word(4 bytes)
- `$s3` contains the base address (address of first element, `A[0]`) of array A. Variable `h` is mapped to `$s2`

4.2.4 Memory Instructions: Others

- Other than load word (`lw`) and store word (`sw`), there are other variants, example:
  - load byte (`lb`)
  - store byte (`sb`)
- Similar in format:
  - `lb $t1, 12($s3)`
  - `lb $t2, 13($s3)`
- Similar in working except that one byte, instead of one word, is loaded or stored
  - Note that the offset no longer needs to be a multiple of 4
- MIPS disallows loading/storing unaligned word using `lw` / `sw`
  - Pseudo-Instructions *unaligned load word* `ulw` *and unaligned store word* `usw` are provided for this purpose
- Other memory instructions:
  - `lh` and `sh` : load and store halfword
  - `lwl` , `lwr` , `swl` , `swr` : load word left/right, store word left/right

4.2.5 Example: Array



1. **C Statement to translate**: 我们要转换的 C 语句是 `A[3] = h + A[1];` 。这个语句表示要将变量 `h` 与数组 `A` 的第二个元素（索引为1的元素）相加，然后将结果存储在数组 `A` 的第四个元素中（索引为3的元素）。
2. **Variables Mapping**: 这部分为我们提供了 C 语句中变量与 MIPS 寄存器之间的映射关系。即变量 `h` 映射到寄存器 `$s2` ，而数组 `A` 的基地址（第一个元素的地址）映射到寄存器 `$s3` 。

3. **MIPS Instructions**:
    - `lw $t0, 4($s3)`：这条指令从数组 `A` 中加载第二个元素（由于每个元素占 4 字节，所以索引为1的元素的偏移是 4 字节）到临时寄存器 `$t0` 中。
    - `add $t0, $s2, $t0`：这条指令将寄存器 `$s2`（存储变量 `h` 的值）与寄存器 `$t0` 中的值相加，并将结果存储在 `$t0` 中。
    - `sw $t0, 12($s3)`：这条指令将 `$t0` 中的值存储到数组 `A` 的第四个元素（由于每个元素占 4 字节，所以索引为3的元素的偏移是 12 字节）。
4. **Memory Representation**：这部分展示了数组 `A` 在内存中的表示方式。从基地址 `$s3` 开始，每个格子表示数组的一个元素。每个元素都是一个 word，这里假设一个 word 的大小是 4 字节。

4.2.6 Common Questions

Address vs Value

> Registers do NOT have types

- A register can hold any 32-bit number:
    - The number has no implicit data type and is interpreted according to the instruction that use it
- Examples:
    - `add $t2, $t1, $t0`
        - `$t0` and `$t1` should contain data values
    - `lw $t2, 0($t0)`
        - `$t0` should contain a memory address

Byte vs Word

> Consecutive word addresses in machines with byte-addressing do not differ by 1

- Common error:
    - Assume that the address of the next word can be found by incrementing the address in a register by 1 instead of by the word size in bytes
- For both `lw` and `sw`:
    - The sum of base address and offset must be a multiple of 4 (i.e. to adhere to word boundary)

4.2.7 Example: Swapping Elements

C Statement to translate	Variables Mapping

`swap( int v[], int k )`
`{`
`  int temp;`
`  temp = v[k]`
`  v[k] = v[k+1];`
`  v[k+1] = temp;`
`}`
	k ➔ **$5**
	Base address of **v[]** ➔ **$4**
	temp ➔ **$15**

Example: k = 3; to swap v[3] with v[4].
Assume base address of v is 2000.

$5 (k) ← 3
$4 (base addr. of v) ← 2000

```
swap:
    sll   $2, $5, 2
    add   $2, $4, $2
    lw    $15, 0($2)
    lw    $16, 4($2)
    sw    $16, 0($2)
    sw    $15, 4($2)
```

$2 ← 12
$2 ← 2012

$15 ← content of mem. addr. 2012 (v[3])
$16 ← content of mem. addr. 2016 (v[4])

content of mem. addr. 2012 (v[3]) ← **$16**
content of mem. addr. 2016 (v[4]) ← **$15**

交换数组中两个连续元素的值（C和MIPS）

- 我们要转换的C函数是swap，函数的主要逻辑是交换数组 `v` 中索引为 `k` 和 `k+1` 的两个连续元素。
- **Variables Mapping**：这部分为我们提供了 C 函数中变量与 MIPS 寄存器之间的映射关系。即参数 `k` 映射到寄存器 `$5`，数组 `v` 的基地址映射到寄存器 `$4`，局部变量 `temp` 映射到寄存器 `$15`。
- **Example**：提供了一个具体的例子，即当 `k=3` 时，交换数组 `v` 的第四和第五个元素（`v[3]` 和 `v[4]`）。假设数组的基地址是 2000。
- **MIPS Instructions**：
  - `sll $2, $5, 2`：这条指令是将 `k`（存储在 `$5` 中）乘以 4（因为每个整数大小是4字节），结果保存在 `$2`。这是为了计算数组中索引为 `k` 的元素的偏移量。
  - `add $2, $4, $2`：将基地址（存储在 `$4`）和偏移量（存储在 `$2`）相加，计算出 `v[k]` 的地址，并将其保存在 `$2` 中。
  - `lw $15, 0($2)`：加载 `v[k]` 的值到 `$15`。
  - `lw $16, 4($2)`：加载 `v[k+1]` 的值到 `$16`。
  - `sw $16, 0($2)`：将 `v[k+1]` 的值存储到 `v[k]` 的位置。
  - `sw $15, 4($2)`：将 `v[k]` 的值存储到 `v[k+1]` 的位置。

4.3 Making Decisions

- Decision make in high-level language:
  - `if` and `goto` statement
  - MIPS decision making instructions are similar to `if` statement with a `goto`
- Decision making instructions

  - Alter the control flow of the program
  - Change the next instruction to be executed
- Two types of decision-making statements in MIPS
  - Conditional (branch)
    `bne $t0, $t1, label`
    `beq $t0, $t1, label`
  - Unconditional (jump)
    `j label`
- A label is an anchor in the assembly code to indicate point of interest, usually as branch target
  - Labels are NOT instructions

1. **Decision make in high-level language:**
   - 当我们在高级编程语言（如 C、Java 或 Python）中进行决策时，通常使用的是 `if` 语句和 `goto` 语句。
   - MIPS 的决策制定指令与高级编程语言中的 `if` 语句相似，并经常与 `goto` 语句一起使用，以决定程序的执行流程。
2. **Decision making instructions:**
   - 这些指令用于改变程序的控制流程。
   - 它们会改变下一条要执行的指令，从而使程序可能跳转到不同的部分执行。

4.3.1 Conditional Branch: `beq` and `bne`

- Processor follows the branch only when the condition is satisfied (True)
- `beq $r1, $r2, L1`
  - Go to statement labeled `L1` if the value in register `$r1` equals the value in register `$r2`
  - `beq` is “branch if equal”
  - C code: `if (a==b) goto L1`
- `bne $r1, $r2, L1`
  - Go to statement labeled `L1` if the value in register `$r1` does not equal the value in register `$r2`
  - `bne` is “branch if not equal”
  - C code: `if (a ≠ b) goto L1`

4.3.2 Unconditional Jump: `j`

- Processor always follows the branch
- `j L1`
  - Jump to label `L1` unconditionally
  - C code: `goto L1`

- Technically equivalent to such statement
  `beq $s0, $s0, L1`

4.3.3 IF statement

C Statement to translate	Variables Mapping
`if (i == j)` `  f = g + h;`	f → $s0 g → $s1 h → $s2 i → $s3 j → $s4

```
      beq $s3, $s4, L1
      j   Exit
L1:   add $s0, $s1, $s2
Exit:
```

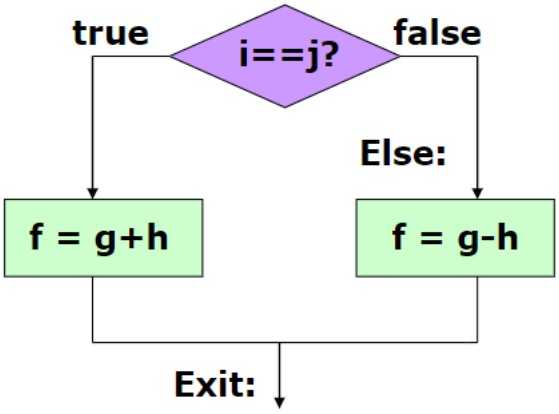
```
      bne $s3, $s4, Exit
      add $s0, $s1, $s2
Exit:
```

The right one is more efficient

C Statement to translate	Variables Mapping

`if (i == j)`
`  f = g + h;`
`else`
`  f = g - h;`
	f → $s0
	g → $s1
	h → $s2
	i → $s3
	j → $s4

```
      bne $s3, $s4, Else
      add $s0, $s1, $s2
      j   Exit
Else: sub $s0, $s1, $s2
Exit:
```



Re-write to `beq`

```
1 beq $s3, $s4, Else
2 sub $s0, $s1, $s2
3 j Exit
4 Else: add $s0, $s1, $s2
5 Exit:
```

4.3.4 Exercise #1: IF statement

MIPS code to translate into C	Variables Mapping

`beq $s1, $s2, Exit`
`add $s0, $zero, $zero`
`Exit:`
	f → $s0
	i → $s1
	j → $s2

- What is the corresponding high-level statement?

```
if (i != j) {
   f = 0;
}
```

4.4 Loops

- C while-loop:

```
while (j == k)
   i = i + 1;
```

- Rewritten with goto

```
Loop:  if (j != k)
          goto Exit;
       i = i+1;
       goto Loop;

Exit:
```

Key concept:

Any form of loop can be written in assembly with the help of conditional branches and jumps.

4.4.1 Exercise #2: FOR loop

4.4.2 Inequalities

- We have `beq` and `bne`, what about branch-if-less-than?
  - There is no real `blt` instruction in MIPS
- Use `slt` (set on less than) or `slti`

```
1 slt $t0, $s1, $s2
```

```
if ($s1 < $s2)
    $t0 = 1;
else
    $t0 = 0;
```

- To build a `blt $s1, $s2, L` in instruction

```
slt $t0, $s1, $s2
bne $t0, $zero, L
```

```
if ($t1 < $t2)
    goto L;
```

- This is another example of pseudo-instruction
  - Assembler translates (`blt`) instruction in an assembly program into the equivalent MIPS(two) instructions

4.5 Array and Loop

- Typical example of accessing array elements in a loop:

Count the number of zeros in an Array A

- A is word array with 40 elements
- Address of A[] → $t0, Result → $t8

C code:

```
result = 0
i = 0
while (i<40) {
    if (A[i] == 0)
        result++;
    i++
}
```

```
addi $t8, $zero, 0              # Assign variable "result" with value 0
addi $t1, $zero, 0              # Assign variable "i" with value 0
addi $t2, $zero, 160            # Assign a temp anoymous variable with value 160,
point to endpoint of array
loop: bge $t1, $t2, end         # Comparing address
      lw  $t3, 0($t1)           # $t3 ← A[i]
      bne $t3, $zero, skip      # if A[i] ≠ 0, then skip
      addi $t8, $t8, 1          # result++
skip: addi $t1, $t1, 4          # move to the next item
      j loop
end:
```

4.6 Exercises

5 - MIPS Instruction

5.1 Overview and Motivation

- Recap: Assembly instructions will be translated to machine code for actual execution
  - This section shows how to translate MIPS assembly code into binary patterns
- Explains some of the "target facts" from earlier:
  - Why is *immediate* limited to 16 bits
  - Why is *shift* amount only 5 bits

5.2 MIPS Encoding: Basics

- Each MIPS instruction has a fixed-length of 32 bits
  - All relevant information for an operation must be encoded with these bits
- Additional challenge:
  - To reduce the complexity of processor design, the instruction encodings should be as regular as possible
    - Small number of formats, i.e. as few variations as possible

> 每条 MIPS 指令都有固定的32位长度。
>
> - 所有操作必须在这32位内编码。这意味着每条指令，无论其功能如何，都被设计为具有相同的长度。

5.3 MIPS Instruction Classification

- Instructions are classified according to their operands:
  - Instructions with same operand types have same encoding

> R-Format (Register format: `op $r1, $r2, $r3`)
>
> - Instructions which use 2 source registers and 1 destination register
> - e.g. `add`, `sub`, `and`, `or`, `nor`, `slt`, etc
> - Special cases: `srl`, `sll`, etc

> I-Format (Immediate format: `op $r1, $r2, Immd`)
>
> - Instructions which use 1 source register, 1 immediate value and 1 destination register
> - e.g. `addi`, `andi`, `ori`, `slti`, `lw`, `sw`, `beq`, `bne`, etc

> J-Format (Jump Format: `op Immd`)
>
> - `j` instruction uses only one immediate value

1. **按操作数分类的指令**：
    - 同种类型的操作数的指令具有相同的编码格式。这意味着，具有相似操作数结构的指令会按照相同的模式进行编码。
2. **R-Format（寄存器格式）**：
    - 这类指令使用两个源寄存器和一个目标寄存器。
    - 示例：`add`，`sub`，`and`，`or`，`nor`，`slt` 等都是 R-格式的指令。
    - 特殊情况：像 `srl` 和 `sll` 这样的位移指令也是 R-格式的指令，但它们有些特别，因为它们可能涉及到一个立即数值（位移的数量）。
3. **I-Format（立即数格式）**：
    - 这类指令使用一个源寄存器，一个立即数值（通常是一个具体的数值，而不是另一个寄存器的值），以及一个目标寄存器。
    - 示例：`addi`，`andi`，`ori`，`slti`（这些都是与常数值进行操作的算术和逻辑指令），以及 `lw`，`sw`（加载和存储指令），`beq`，`bne`（分支指令）等都是 I-格式的指令。
4. **J-Format（跳转格式）**：
    - 这类指令主要与跳转操作有关。
    - `j` 指令就是一个示例，它只使用一个立即数值来表示跳转的目标地址。

5.4 MIPS Registers

- For simplicity, register numbers (`$0, $1, ..., $31`) will be used in examples here instead of register names

Name	Register number	Usage
$zero	0	Constant value 0
$v0-$v1	2-3	Values for results and expression evaluation
$a0-$a3	4-7	Arguments
$t0-$t7	8-15	Temporaries
$s0-$s7	16-23	Program variables

Name	Register number	Usage
$t8-$t9	24-25	More temporaries
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address

5.5 R-Format

- Define fields with the following number of bits each:
    - 6+5+5+5+5+6=32 bits

6	5	5	5	5	6

- Each field has a name:

opcode	rs	rt	rd	shamt	funct

- Each field is an independent 5 or 6 bits unsigned integer
  - A 5-bit field can represent any number 0-31
  - A 6-bit field can represent any number 0-63

Fields	Meaning
opcode	- Partially specifies the instruction - Equal to 0 for all R-Format instructions
funct	- Combined with opcode exactly specifies the instruction
rs (Source Register)	- Specify register containing first operand
rt (Target Register)	- Specify register containing second operand
rd (Destination Register)	- Specify register which will receive result of computation
shamt	- Amount a shift instruction will shift by - 5 bits (i.e. 0 to 31) - Set to 0 in all non-shift instructions

MIPS R-Format 指令用于表示那些涉及两个源寄存器和一个目的地寄存器的指令，如加法、减法和位逻辑操作等。这些字段将 32 位的指令分解为特定的部分，每个部分有其特定的功能和意义。

以下是对每个字段的解释：

1. **opcode**：操作码字段，用于部分指定指令的类型。对于所有的 R-Format 指令，opcode 都设置为 0。
2. **funct**：函数代码字段，与 opcode 一起合并，可以准确地指定指令的类型（例如，加法、减法等）。
3. **rs (Source Register)**：源寄存器字段，指定第一个操作数的寄存器。
4. **rt (Target Register)**：目标寄存器字段，指定第二个操作数的寄存器。
5. **rd (Destination Register)**：目的地寄存器字段，指定存放操作结果的寄存器。
6. **shamt**：位移量字段，用于指定位移指令（如 srl 和 sll）的位移量。此字段有 5 位，因此可以表示从 0 到 31 的值。对于非位移指令，此字段设置为 0。

总体来说，这些字段组合在一起，形成了一个 32 位的 R-Format 指令，允许处理器知道它需要执行什么操作以及操作的操作数是哪些。这些字段是指令编码的基础，使得处理器能够快速地解码和执行这些指令。

5.5.1 R-Format: Example

MIPS instruction

```
add  $8, $9, $10
```

R-Format Fields	Value	Remarks
opcode	0	(textbook pg 94 - 101)
funct	32	(textbook pg 94 - 101)
rd	8	(destination register)
rs	9	(first operand)
rt	10	(second operand)
shamt	0	(not a shift instruction)

MIPS instruction

```
add  $8, $9, $10
```

Note the ordering of the 3 registers

Field representation in decimal:

opcode	rs	rt	rd	shamt	funct
0	9	10	8	0	32

Field representation in binary:

000000	01001	01010	01000	00000	100000

Split into 4-bit groups for hexadecimal conversion:

0000	0001	0010	1010	0100	0000	0010	0000
0_{16}	1_{16}	2_{16}	A_{16}	4_{16}	0_{16}	2_{16}	0_{16}

MIPS instruction

```
sll  $8, $9, 4
```

Note the placement of the source register

Field representation in decimal:

opcode	rs	rt	rd	shamt	funct
0	0	9	8	4	0

Field representation in binary:

000000	00000	01001	01000	00100	000000

Split into 4-bit groups for hexadecimal conversion:

0000	0000	0000	1001	0100	0001	0000	0000
0_{16}	0_{16}	0_{16}	9_{16}	4_{16}	1_{16}	0_{16}	0_{16}

5.6 I-Format

- 5-bit `shamt` field can only represent 0 to 31
- Immediates may be much larger than this
  - e.g. `lw`, `sw` instructions require bigger offset
- Compromise: Define a new instruction format partially consistent with R-format
  - If instruction has immediate, then it uses at most 2 registers
- Define fields with the following number of bits each:

6	5	5	16

opcode	rs	rt	immediate

- I-Format has no `funct` field, so the `opcode` uniquely specifies an instruction
- `rs` field specifies the source register operand
- `rt` field specifies the register to receive the result
- Immediate treated as a **single integer**

5.6.1 Instruction Address: Overview

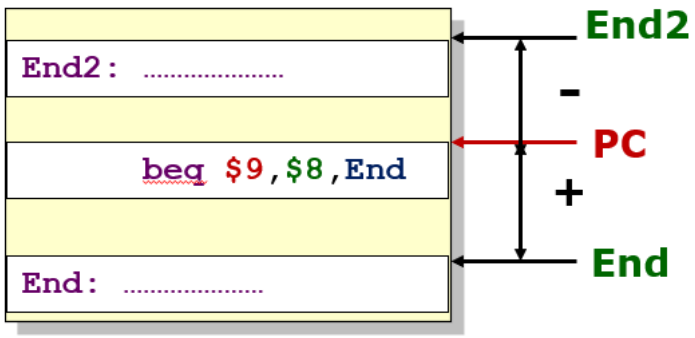
- As instructions are stored in memory, they too have addresses
  - Conrtol flow instructions uses these addresses
- As instructions are 32-bit long, instruction addresses are word-aligned as well
- Program Counter(PC) is a special register that keeps address of instruction being executed in the processor

在介绍I-format指令时引入program counter (PC) 是因为一些I-format指令，特别是那些涉及分支和跳转的指令，使用立即数值作为基于当前PC值的偏移。这样的偏移方式允许指令跳转到程序内的特定位置。

5.6.2 Branch: PC-Relative Addressing

- Use I-Format

- `opcode` specifies `beq`, `bne`
- `rs` and `rt` specify registers to compare
- `immediate` is only 16 bits, the memory address is 32 bits. `immediate` is not enough to specify the entire target address
- Solution: Specify target address relative to the PC
- Target address is generated as :
  - PC + 16-bit `immediate` field
  - The `immediate` field is a signed two's complement integer



相对寻址：许多I-format指令，如 `beq` (branch if equal) 和 `bne` (branch if not equal)，使用立即字段作为跳转偏移量。这些偏移量是相对于当前的PC值的。通过这种方法，指令可以确定跳转到程序中的哪个位置。

1. **`immediate` is only 16 bits**：在I-format指令中，

立即数字段（immediate field）的大小是固定的16位。这意味着它可以直接表示的值的范围是从0到65535（如果被视为无符号数）或从-32768到32767（如果被视为有符号数）。

1. **the memory address is 32 bits**：在MIPS架构中，完整的内存地址是32位的。这意味着系统可以直接访问的内存地址范围是从0到4GB。

2. **`immediate` is not enough to specify the entire target address**：由于立即数字段只有16位，它不能直接表示32位的完整内存地址。换句话说，一个16位的值不足以覆盖整个4GB的内存空间。

这有什么意义呢?

当使用I-format指令（如加载和存储指令）访问内存时，通常使用一个基址寄存器和一个16位的偏移量（即立即数）来确定目标地址。例如，在 `lw $t0, 4($t1)` 这样的指令中，`$t1` 包含一个基地址，而4是一个16位的偏移量。最终的内存访问地址是 `$t1` 中的值加上4。

此外，对于跳转和分支指令，16位的立即数经常被解释为相对于当前指令地址（即程序计数器，PC）的偏移量。这允许指令跳转到相对近的位置，而不需要指定完整的32位地址。

综上所述，尽管16位的立即数不能直接指定完整的32位地址，但通过与其他值（如基址寄存器或当前PC值）结合，可以有效地指定或计算目标地址。

Branches

- We usually use branches by `if-else`, `while`, `for`
- Loops are generally small:
  - Typically up to 50 instructions
- Unconditional jumps are done using jump instruction `j`, not the branches
- Conclusion: A branch often changes PC by a small amount
- Can the branch target range be enlarged?
- Observation: Instructions are word-aligned
  - Number of bytes to add to the PC will always be a multiple of 4 (Since each MIPS instruction is 32-bit)
  - Interpret the `immediate` as a number of words, i.e. automatically multiplied by 4_{10}(100_2)
- Can branch to $\pm 2^{15}$ words from the PC
  - i.e. $\pm 2^{17}$ bytes from the PC

Branch Calculation

If the branch is **not taken**:

$$PC = PC + 4$$

(**PC + 4** is address of next instruction)

If the branch is **taken**:

$$PC = (PC + 4) + (immediate \times 4)$$

- `immediate` field specifies the number of words to jump, which is the same as the number of instructions to 'skip over'
- `immediate` field can be positive or negative

- Due to hardware design, add `immediate` to (PC+4), not to PC

即如果没有进入branch，则跳转到下一个指令，由于每个MIPS指令是32 bits，所以在PC上加4字节

如果进入了branch，则跳转到一个新的地址来执行指令，这个新的地址是基于当前PC值，偏移量（即下一条指令的位置PC+4），以及由分支指令中的立即字段（ `immediate` ）给出的额外偏移量计算出来的

立即数字段（ `immediate` ）被解释为word的数量，而每个word有4字节，所以被乘以4

5.7 J-Format

- For branches, PC-relative addressing was used, because we do not need to branch too far
- For general jump `j`, we may jump to anywhere in memory
- The ideal case is to specify a 32-bit memory address to jump to.

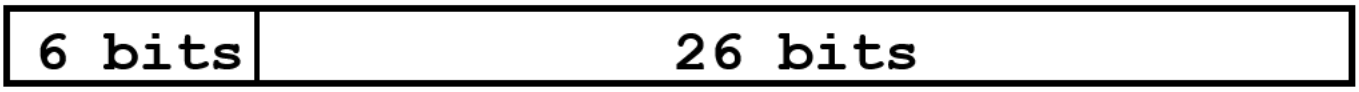
J-format 是 MIPS 指令集中用于跳转（jump）指令的格式。虽然理论上我们确实希望直接指明一个32位的内存地址以进行跳转，但在实际的指令编码中，由于指令的长度也是32位，不可能在单一的指令中同时包含操作码、其他字段和一个完整的32位地址。

考虑到这个限制，J-format 被设计为只包含一个26位的跳转目标地址字段。这意味着我们不能直接指定整个32位的跳转目标地址。但是，由于所有指令都是字对齐的，这意味着指令的地址的最后两位总是00（因为指令长度为4字节，或32位）。这为我们提供了一个小小的优势：我们只需要指定高26位的地址，而低2位则可以简单地置为0。

此外，J-format 的跳转是基于绝对地址的，而不是相对于当前 PC 的偏移。这意味着 J-format 跳转指令可以跳转到内存中的任何位置，只要这个位置在那26位地址范围内。

总的来说，虽然我们希望能够直接指明一个32位的地址进行跳转，但由于指令长度的限制和字对齐的特性，J-format 只提供了一个26位的地址字段来进行跳转。

- Define fields of the following number of bits each

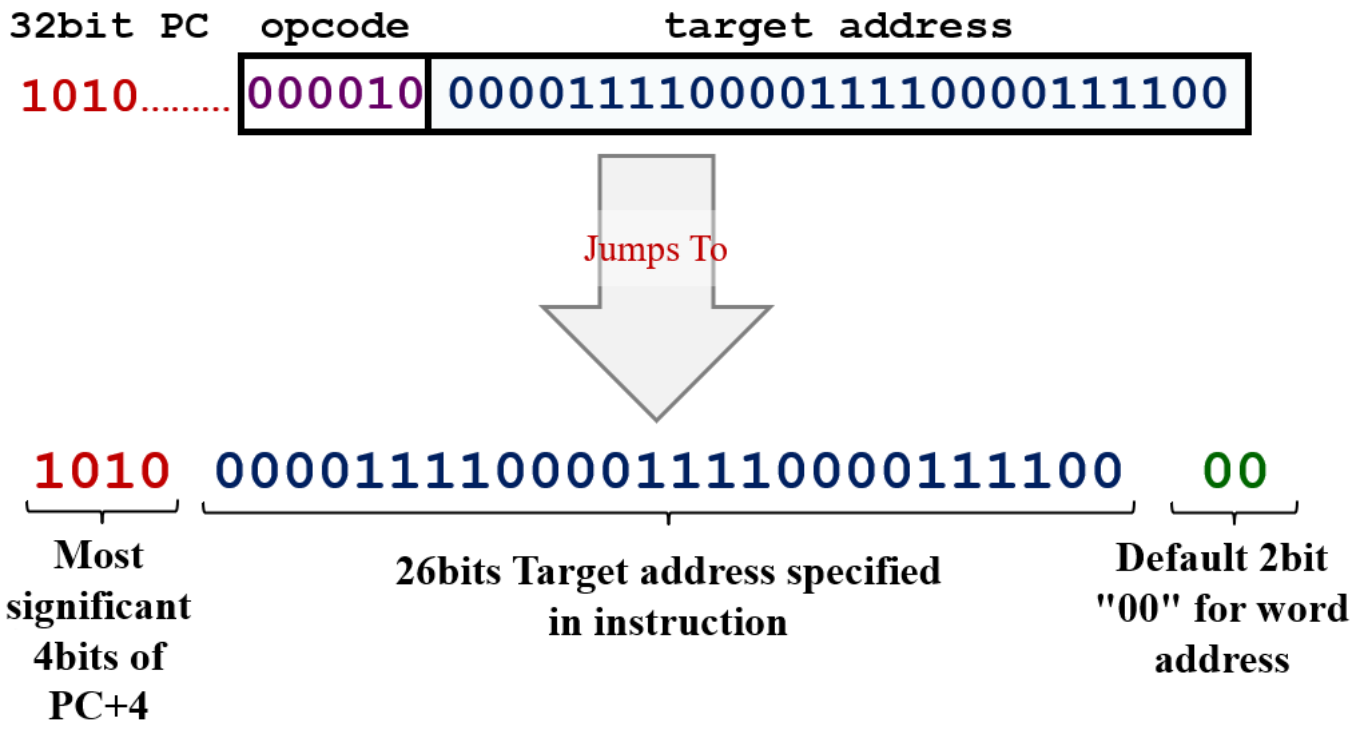


- Keep `opcode` field identical to R-format and I-format for consistency
- Combine all other fields to make room for larger target address
- We can only specify 26-bits of 32 bits address
- Optimization:
  - Just like with branches, jumps will only jump to word-aligned addresses, so last two bits are always `00`
  - So, let's assume the address ends with `00` and leave them out
  - Now we can specify **28 bits** of 32-bit address

- Where do we get the other 4 bits?
  - MIPS choose to take the 4 most significant bits from PC+4
  - This means that we cannnot jump to anywhere in memory, but it should be sufficient most of the time
  - The maximum jump range is **256MB**
- Special instruction if the program straddles 256MB boundary
  - `jr`: use `load` instruction to load full 32-bit memory address to register, then use `jr` instruction to jump
  - Target address is specified through a register

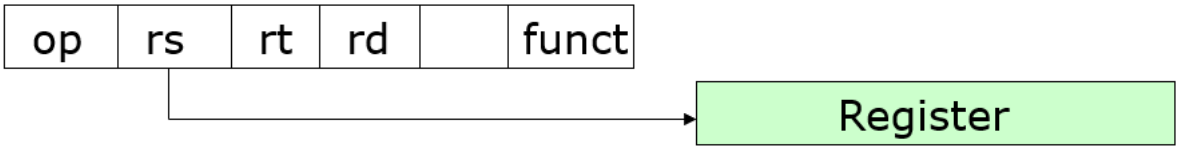
> 在MIPS中，使用J-format的 `j` 指令时，我们确实只有26位来指定跳转地址。这26位的地址是如何转化为完整的32位地址的呢？它是通过将这26位的跳转字段左移2位（因为每个指令是4字节或32位）来实现的，然后再与当前指令地址的高4位进行组合，从而生成完整的32位跳转地址。
>
> 因此，这26位地址可以指定 2^{26} 不同的跳转目标，也就是 67,108,864 个可能的目标。由于每个指令都是4字节，所以跳转范围是 $2^{26} * 4$ 字节，也就是 268,435,456 字节或 256 MB。

Summary: Given a Jump instruction



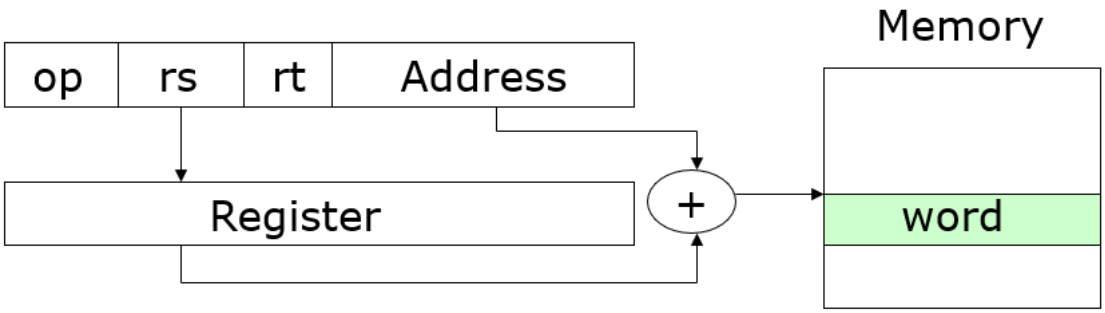
5.8 Addressing Modes

- **Register addressing**: operand is a register

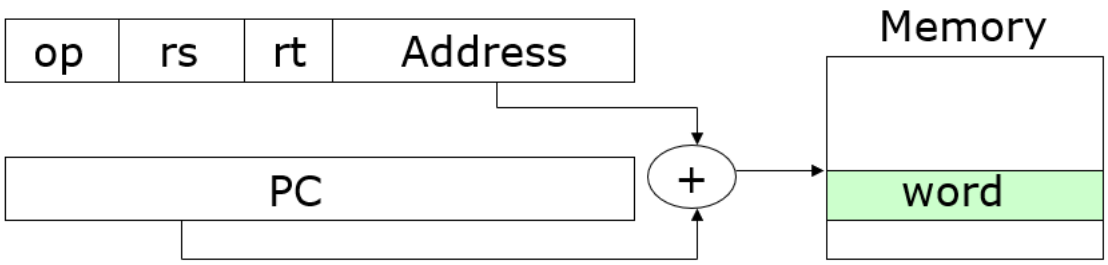


- Immediate addressing: operand is a constant within the instruction itself (`addi` , `andi` , `ori` , `slti`)

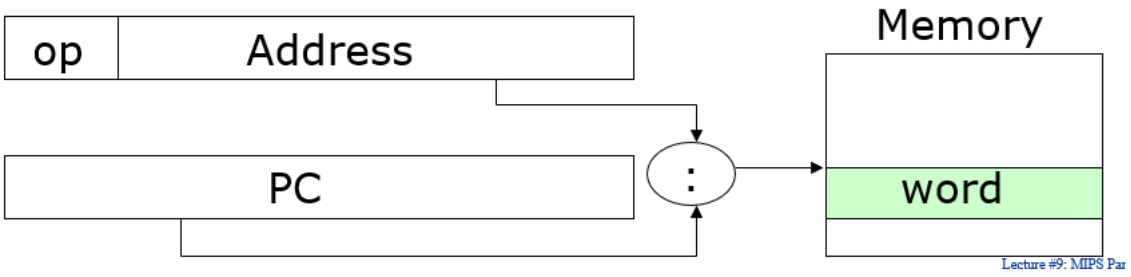
- Base addressing (displacement addressing): operand is at the memory location whose address is sum of a register and a constant in the instruction (`lw` , `sw`)



- PC-relative addressing: address is sum of PC and constant in the instruction (`beq` , `bne`)



- Pseudo-direct addressing: 26-bit of instruction concatenated with upper 4-bits of PC (`j`)



5.9 Summary

- MIPS instruction:
  - 32 bits representing a single instruction, for each format (R, I, J) the fields included are different

R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	immediate		
J	opcode	target address				

- Branches and load/store are both I-format instructions; but branches use PC-relative addressing, whereas load/store use base addressing
- Branches use PC-relative addressing; jumps use pseudo-direct addressing
- Shifts use R-format, but other immediate instructions (`addi` , `andi` , `ori`) use I-format

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
Arithmetic	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
	load w ord	`lw $s1, 100($s2)`	`$s1 = Memory[$s2 + 100]`	Word from memory to register
	store w ord	`sw $s1, 100($s2)`	`Memory[$s2 + 100] = $s1`	Word from register to memory
Data transfer	load byte	`lb $s1, 100($s2)`	`$s1 = Memory[$s2 + 100]`	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	`Memory[$s2 + 100] = $s1`	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
Conditional branch	set on less than	`slt $s1, $s2, $s3`	if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0	Compare less than constant
	jump	`j 2500`	go to 10000	Jump to target address
Uncondi-	jump register	`jr $ra`	go to `$ra`	For sw itch, procedure return
tional jump	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

6 - Datapath Design

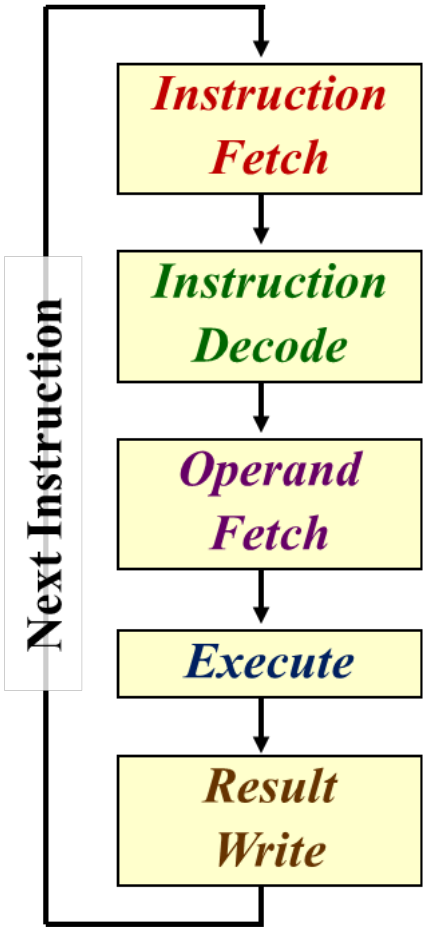
6.1 Building a Processor: Datapath & Control

- Two major components for a processor:
  1. Datapath
     - Collection of components that process data
     - Performs the arithmetic, logical and memory operations
  2. Control
     - Tells the datapath, memory and I/O devices what to do according to program instructions

6.2 MIPS Processor: Implementation

- Simplest possible implementation of a subset of the core MIPS ISA:
  - Arithmetic and Logical operations
    - `add`, `sub`, `and`, `or`, `addi`, `andi`, `ori`, `slt`
  - Data transfer instructions
    - `lw`, `sw`
  - Branches
    - `beq`, `bne`
- Shift instructions (`sll`, `srl`) and J-type instructions (`j`) will not be discussed

6.3 Instruction Execution Cycle



1. Fetch:
    - Get instruction from memory
    - Address is in Program Counter (PC) Register
2. Decode:
    - Find out the operation required
3. Operand Fetch
    - Get operand(s) needed for operation
4. Execute:
    - Perform the required operation
5. Result Write (Store):
    - Store the result of the operation

6.4 MIPS Instruction Execution

- Show the actual steps for 3 representative MIPS instructions
- Fetch and Decode stage not shown, the standard steps are performed

	`add $3, $1, $2`	`lw $3, 20( $1 )`	`beq $1, $2, label`
Fetch **Decode**	***standard***	***standard***	***standard***
Operand Fetch	○ Read [**$1**] as *opr1* ○ Read [**$2**] as *opr2*	○ Read [**$1**] as *opr1* ○ Use ***20*** as *opr2*	○ Read [**$1**] as *opr1* ○ Read [**$2**] as *opr2*
Execute	*Result* = *opr1* + *opr2*	○ *MemAddr* = *opr1* + *opr2* ○ Use *MemAddr* to read from memory	*Taken* = (*opr1* == *opr2*)? *Target* = (**PC**+4) + **ofst**×**4**
Result Write	*Result* stored in **$3**	*Memory* data stored in **$3**	if (*Taken*) **PC** = *Target*

- `opr` = operand, `ofst` = offset, `MemAddr` = Memory Address
- Design changes:
  - Merge Decode and Operand Fetch - Decode is simple for MIPS
  - Split Execute into ALU (Calculation) and Memory Access

	`add $3, $1, $2`	`lw $3, 20( $1 )`	`beq $1, $2, label`
Fetch	Read inst. at [PC]	Read inst. at [PC]	Read inst. at [PC]
Decode & Operand Fetch	○ Read [**$1**] as *opr1* ○ Read [**$2**] as *opr2*	○ Read [**$1**] as *opr1* ○ Use ***20*** as *opr2*	○ Read [**$1**] as *opr1* ○ Read [**$2**] as *opr2*
ALU	*Result* = *opr1* + *opr2*	*MemAddr* = *opr1* + *opr2*	*Taken* = (*opr1* == *opr2*)? *Target* = (**PC**+4) + **ofst**×**4**
Memory Access		Use *MemAddr* to read from memory	
Result Write	*Result* stored in **$3**	*Memory* data stored in **$3**	if (*Taken*) **PC** = *Target*

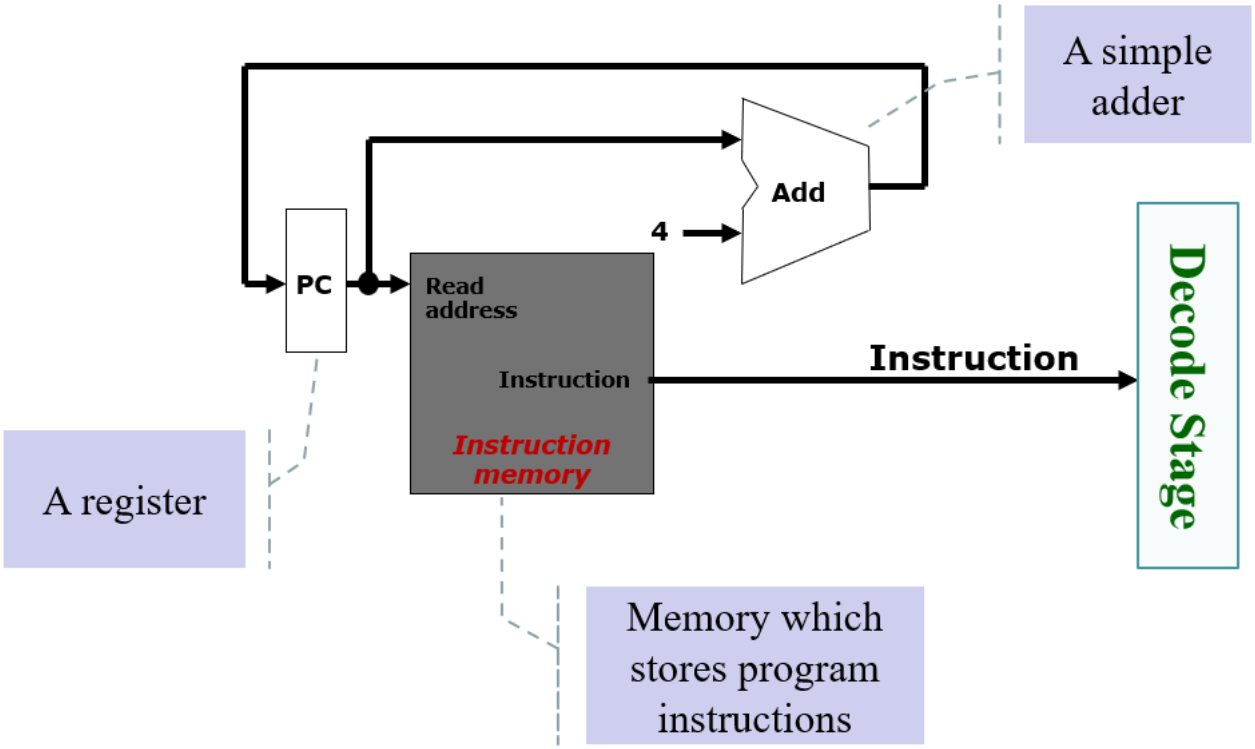
- `inst.` = instruction

6.5 Build a MIPS Processor

- What we are going to do:
  - Look at each stage closely, figure out the requirements and processes
  - Sketch a high level block diagram, then zoom in for each elements
  - With the simple starting design, check whether different type of instructions can be handled
    - Add modifications when needed

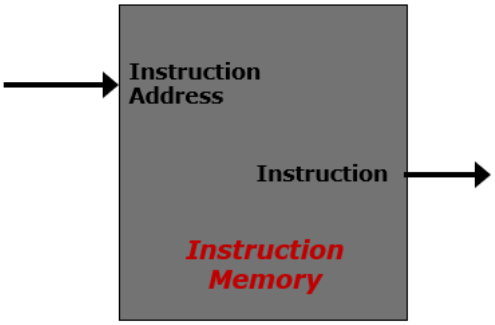
6.5.1 Fetch Stage

- Instruction **Fetch Stage**:
  1. Use the Program Counter (PC) to fetch the instruction from *memory*
     - PC is implemented as a special register in the processor
  2. Increment the PC by 4 to get the address of the next instruction:
     - Since each instruction is 32 bit, which is 4 bytes
     - Note the exception when branch/jump instruction is executed
- Output to the next stage (**Decode**)"
  - The instruction to be executed

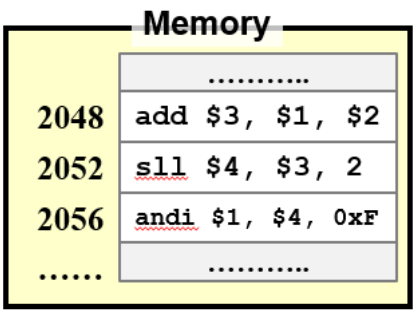


Element: Instruction Memory

- Storage element for the instructions
  - It is a sequential circuit
  - Has an internal state that stores information
  - Clock signal is assumed and not shown



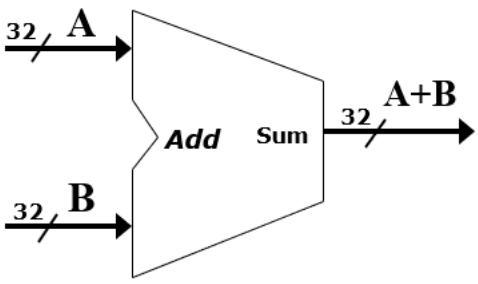
- Supply instruction given the address
  - Given instruction address M as input, the memory outputs the content at address M
  - Conceptual diagram of the memory layout is given on the below



1. **存储元素**：这里提到的存储元素是指用于存储指令的硬件部件。在计算机中，这通常是指主存储器或RAM。
2. **它是一个顺序电路**：与组合电路不同，顺序电路有一定的内部状态，并且其输出不仅取决于当前的输入，还取决于之前的输入或状态。这意味着存储元素可以维持和存储信息。
3. **有内部状态用于存储信息**：这意味着该电路具有某种记忆能力，能够保存数据或状态信息，直到被更改或清除。
4. **时钟信号是假设的并且未显示**：顺序电路通常需要一个时钟信号来同步其操作。但是，在某些描述或图解中，为了简化表示，时钟信号可能不会明确显示。
5. **提供指令给定地址**：存储元素的主要功能之一是，当提供一个指令地址 `M` 时，它能够输出存储在地址 `M` 的内容。这就是计算机从内存中获取指令并执行它们的方式。

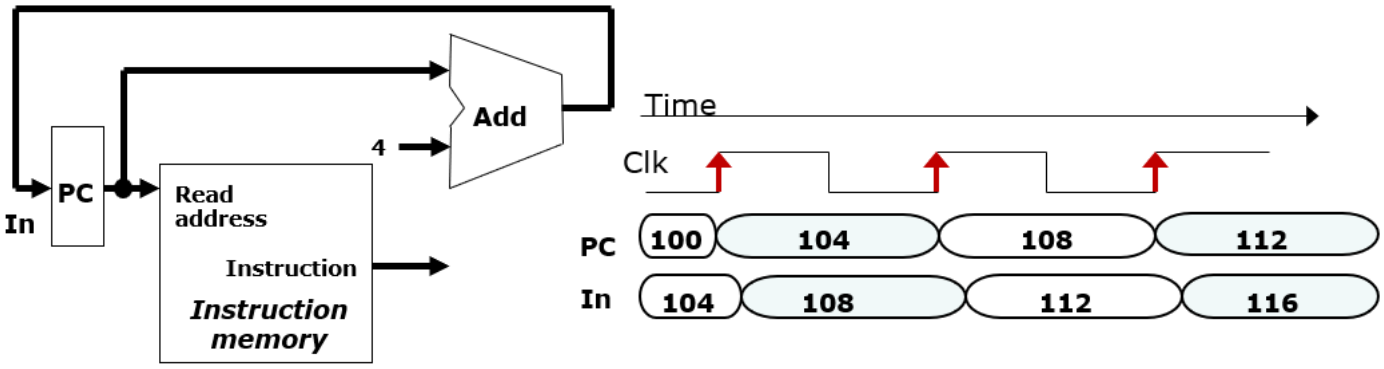
Element: Adder

- Combinational logic to implement the addition of two numbers
- Inputs:
  - Two 32-bit numbers A,B
- Output:
  - Sum of the input number A+B



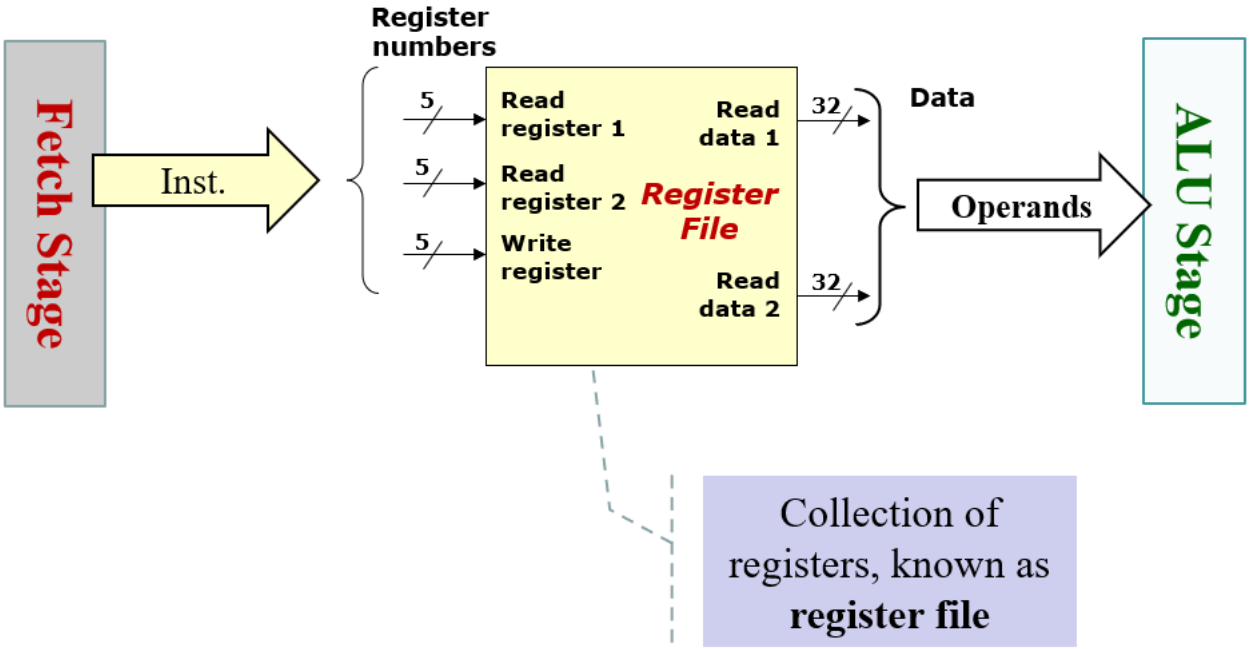
The idea of clocking

- PC is read during the first half of the clock period and it is updated with PC+4 at the next rising clock edge



6.5.2 Decode Stage

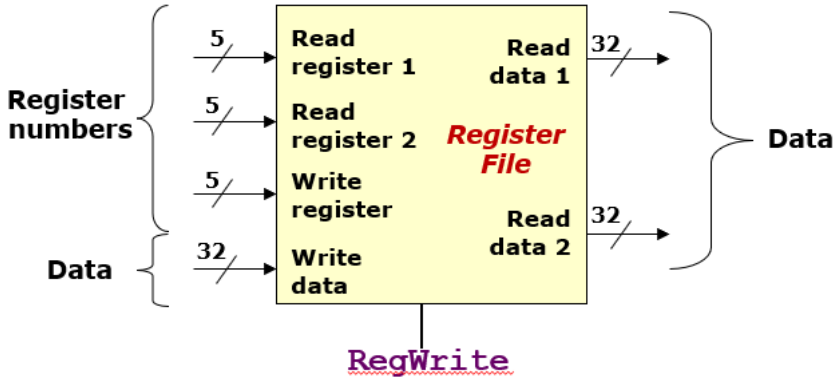
- Instruction Decode Stage:
  - Gather data from the instruction fields:
    1. Read the `opcode` to determine instruction type and field lengths
    2. Read data from all necessary registers
- Input from previous stage (Fetch):
  - Instruction to be executed
- Output to the next stage (ALU):
  - Operation and the necessary operands



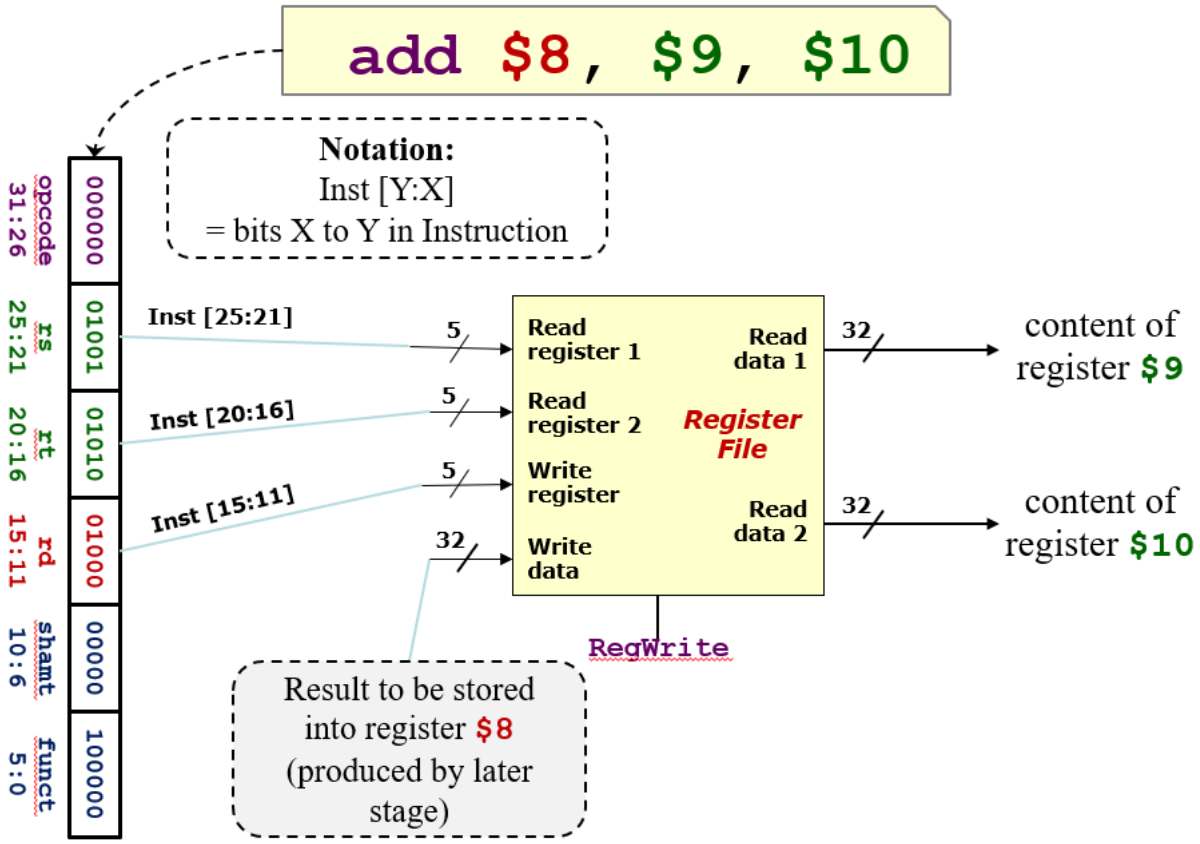
Element: Register File

- A collection of 32 registers
  - Each 32-bit wide; can be read/written by specifying register number
  - Read at most two register per instruction
  - Write at most one register per instruction
- `RegWrite` is a control signal to indicate:

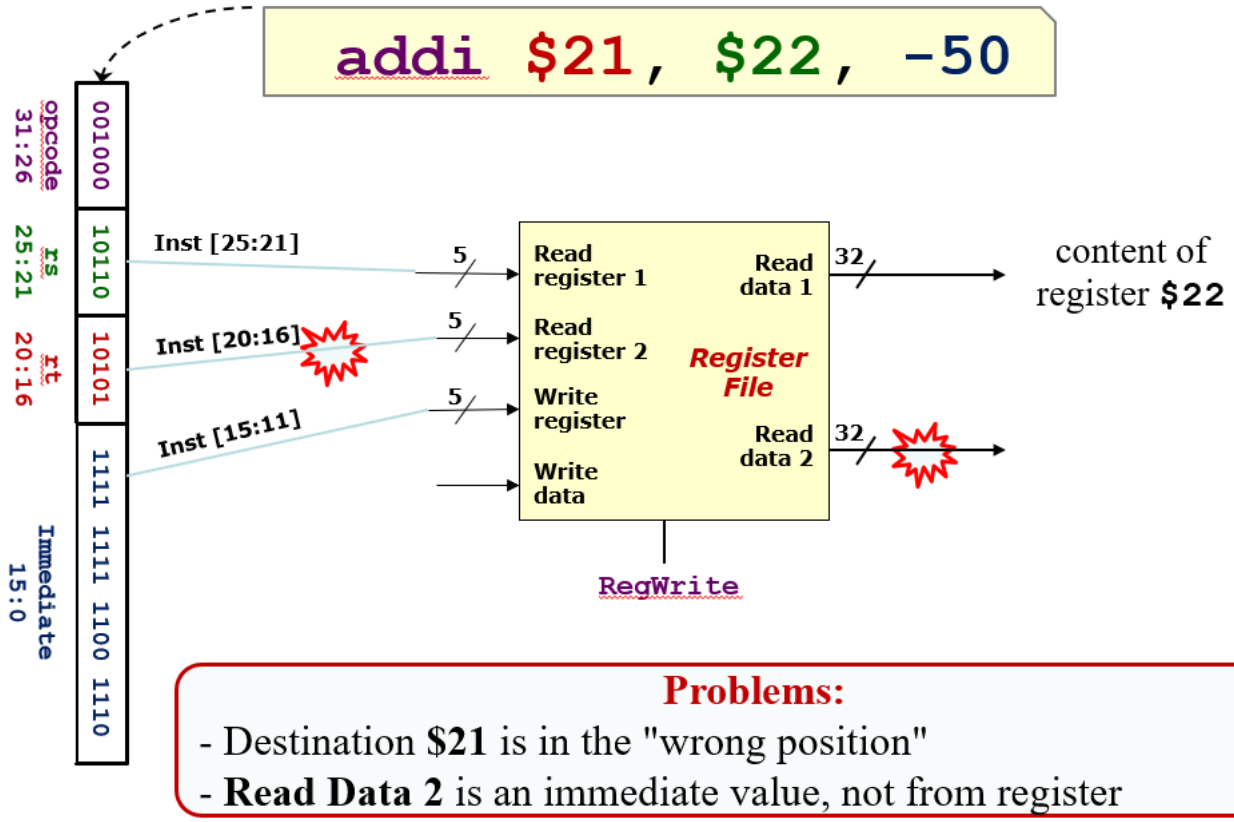
- Writing of register
- 1 (True) = Write, 0 (False) = No Write



R-Format Instruction

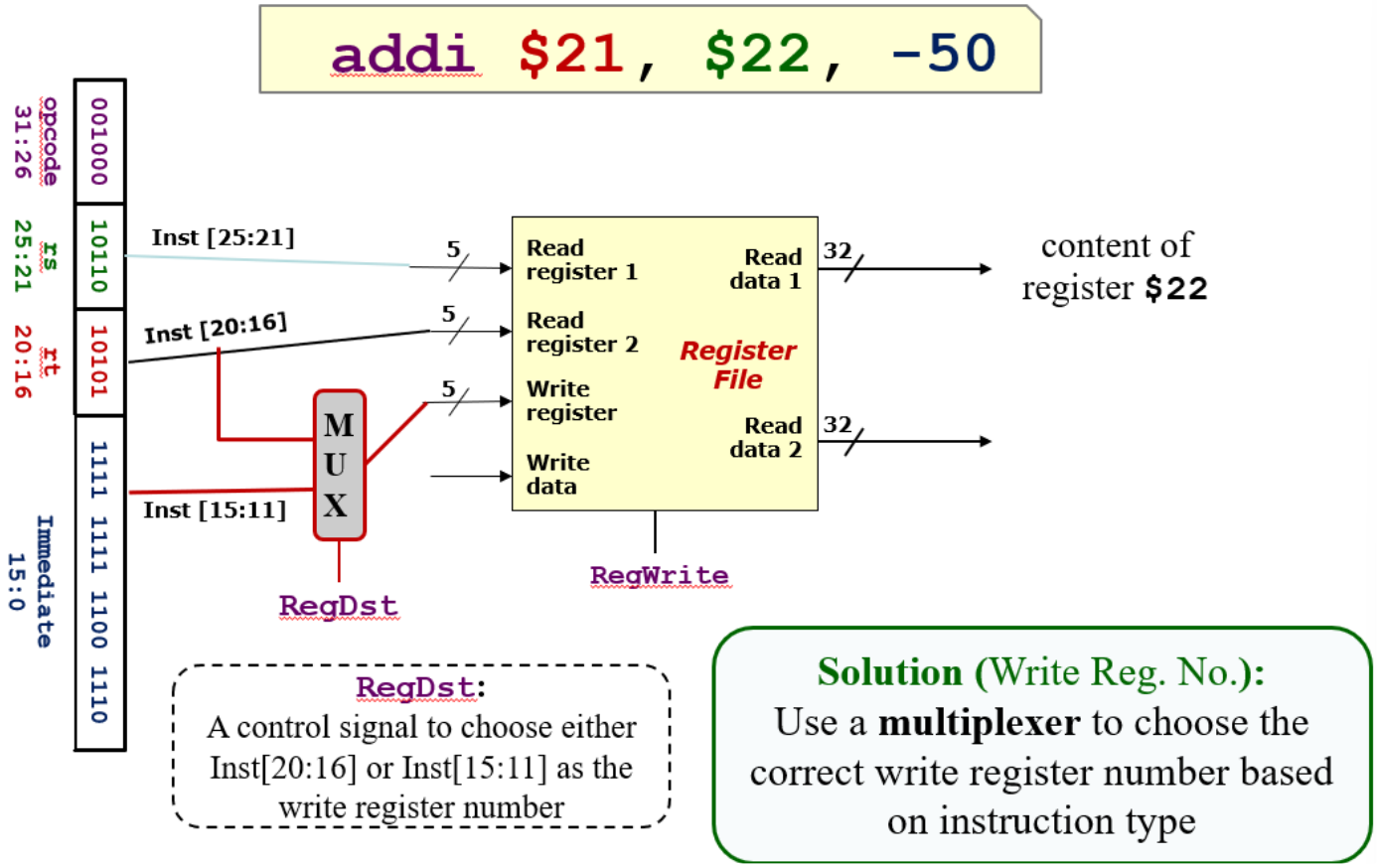


I-Format Instruction



Problems:

- Destination **$21** is in the "wrong position"
- **Read Data 2** is an immediate value, not from register



RegDst:
A control signal to choose either Inst[20:16] or Inst[15:11] as the write register number

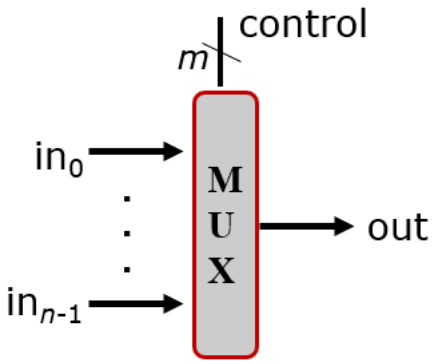
Solution (Write Reg. No.):
Use a **multiplexer** to choose the correct write register number based on instruction type

1. **不同的指令格式**：MIPS指令集包括R-format、I-format和J-format等不同格式的指令。这些指令格式中，寄存器的编号和位置可能会不同。
2. **写入寄存器的选择**：在R-format指令中，通常使用 `rd` 字段（也就是 `Inst[15:11]`）来指定结果的写入寄存器。但是，在I-format指令（如 `addi`）中，结果是写入 `rt` 字段指定的寄存器，也就是 `Inst[20:16]`。

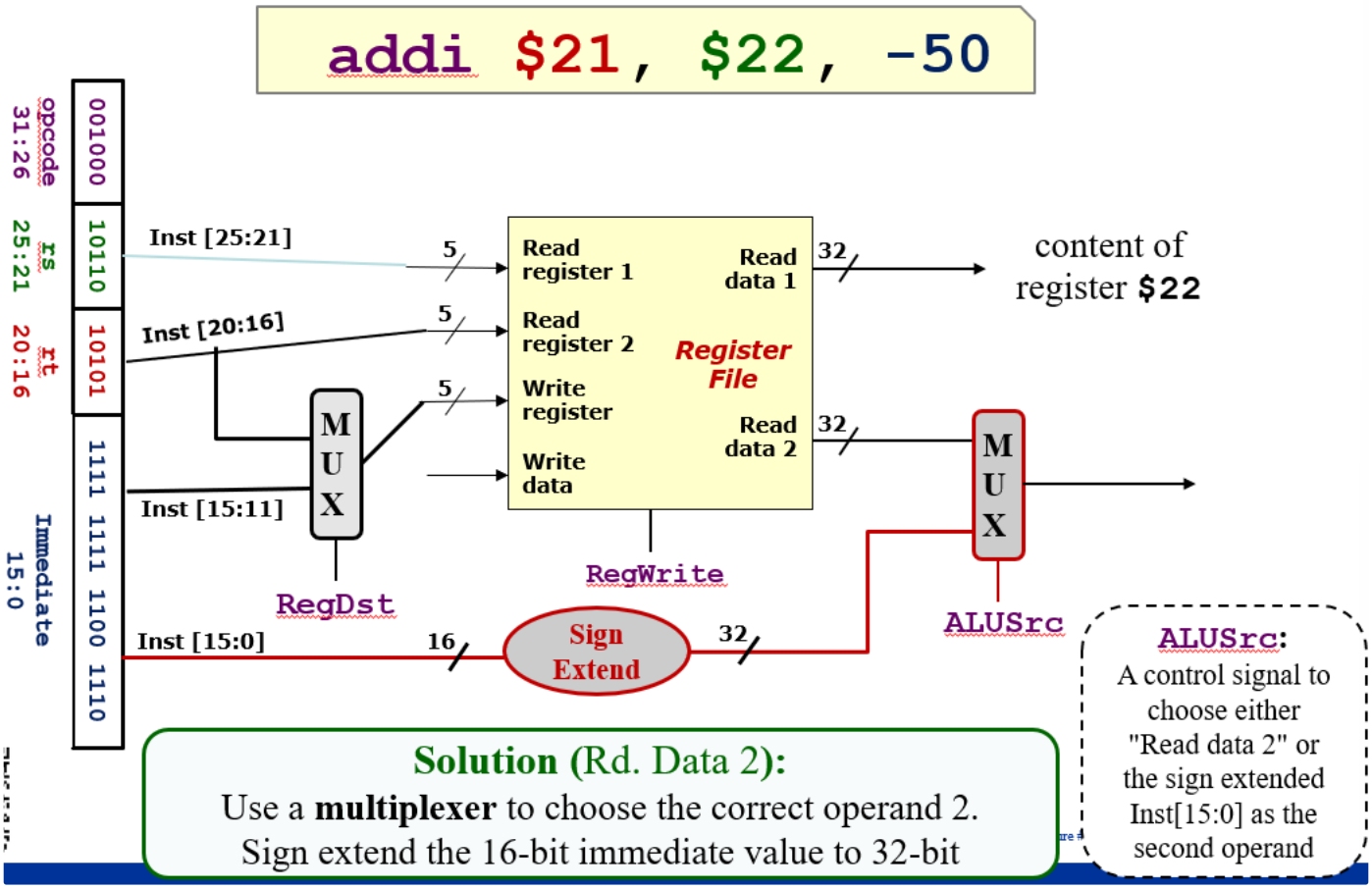
3. **multiplexer的作用**：为了能够处理这种不同，我们需要一个选择器，即 multiplexer 。根据指令的类型（R-format还是I-format）， multiplexer 决定应该使用 Inst[20:16] 还是 Inst[15:11] 来确定写入寄存器的编号。
4. **控制信号 RegDst** ： 这是一个控制信号，用于告诉 multiplexer 该选择哪个输入。例如，对于I-format指令， RegDst 会设置为选择 Inst[20:16] ；对于R-format指令，它会设置为选择 Inst[15:11] 。

Multiplexer

- Function: Selects one input from multiple input lines
- Inputs: n lines of same width
- Control: m bits where $n = 2^m$
- Output: Select i-th input line if control = i



Control=0 → select in_0 to out
Control=3 → select in_3 to out



这张图中显示了完整的decoder处理I-format指令的流程。

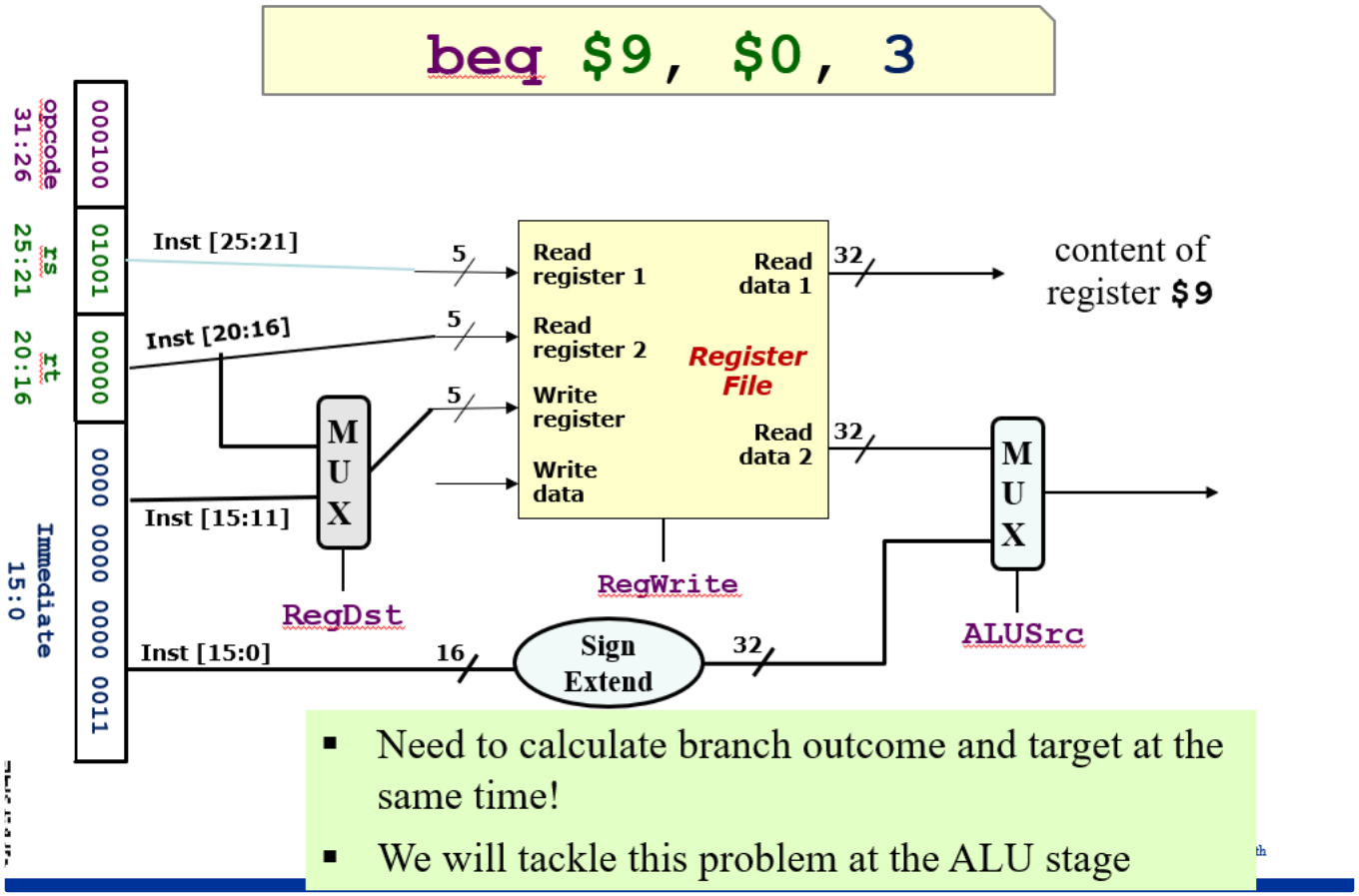
首先在左侧的输入部分，由于I format指令少一个寄存器的输入，所以先加入一个MUX将rt部分和原先属于rd部分的寄存器输入，根据指令的类型选择输出一个到decoder中。最后，对immediate的后半部分（16位）加长到32位后连到data read 2后面的MUX中。

sign extend操作是这样的：首先检查immediate的最高位（第15位，也称为符号位）。如果符号位是0（即该值是正数或零），那么在32位值的前16位都填充0。如果符号位是1（即该值是负数），那么在32位值的前16位都填充1。

由于MIPS的I format指令中的immediate字段一开始就被设计成16位，所以在decode阶段中截断成16位再延长至32位的操作不会导致数据的损失

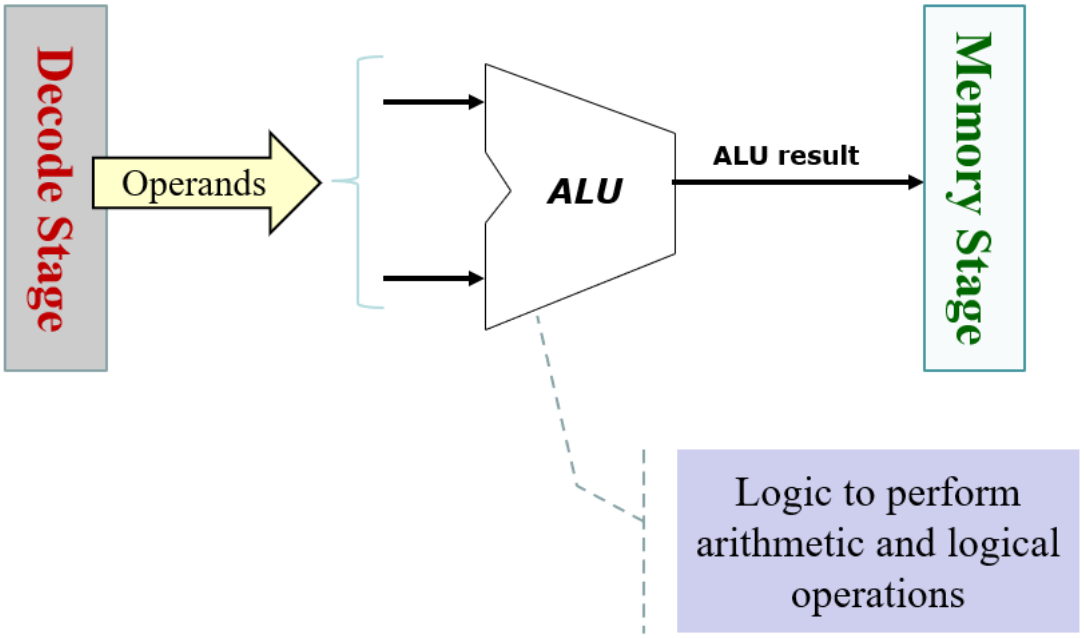
在右半部分，如果指令是I format的，那么左侧的MUX会输入rt，右侧的MUX会选择immediate。最终decode阶段输出的是一个register值和一个immediate值

Branch Instruction



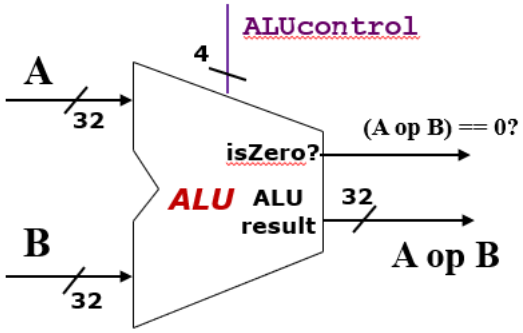
6.5.3 ALU Stage

- Instruction ALU stage:
  - ALU = Arithmetic Logic Unit
  - Also called the Execution stage
  - Perform the real work for most instruction here
    - Arithmetic (e.g. `add`, `sub`), shifting(`sll`), Logical (`and`, `or`)
    - Memory operation (`lw`, `sw`): Address calculation
    - Branch operation (`bne`, `beq`): Perform register comparison and target address calculation
- Input from previous stage (Decode):
  - Operations and operands
- Output to the next stage (Memory):
  - Calculation result



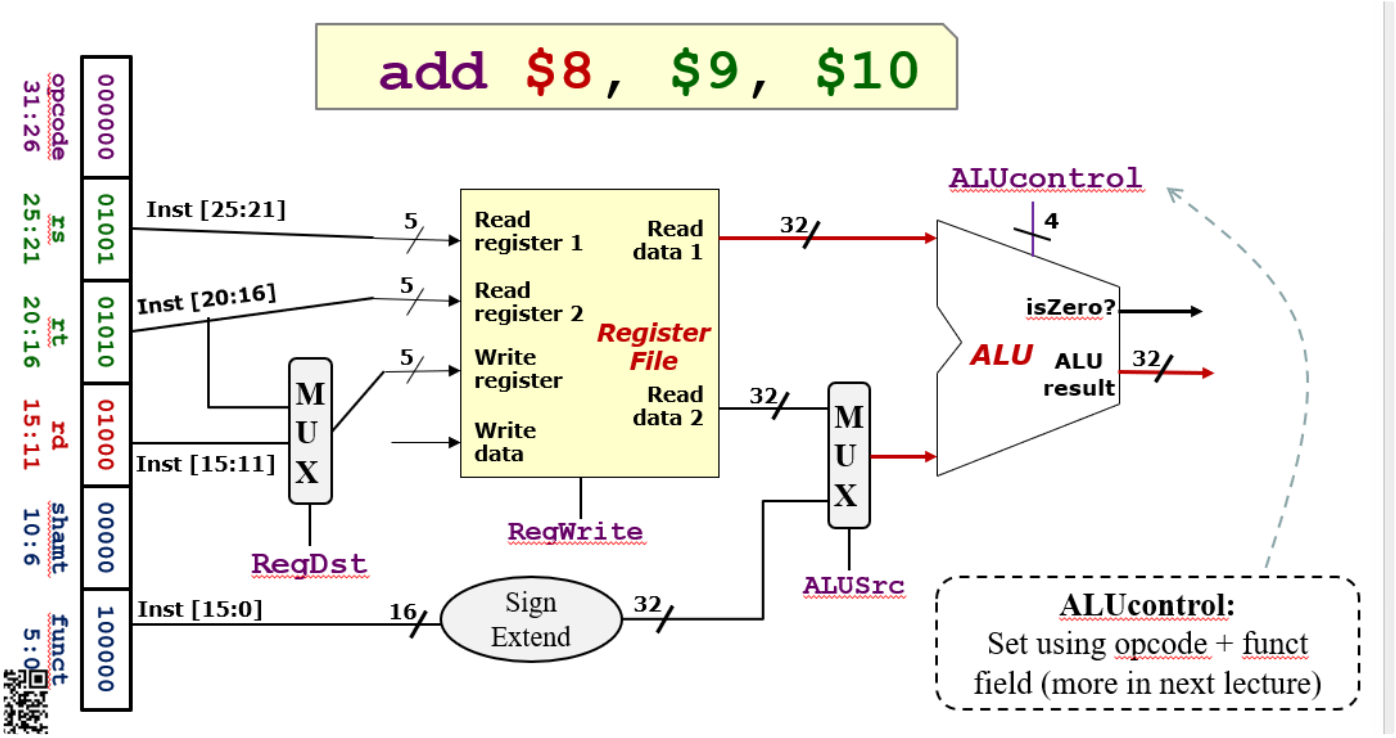
Element: Arithmetic Logic Unit

- ALU is a combinational logic to implement arithmetic and logical operations
- Inputs: two 32-bit numbers
- Control: 4-bit to decide the particular operation
- Outputs: Result of arithmetic/logical operation, and a 1-bit signal to indicate whether the result is zero



ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR

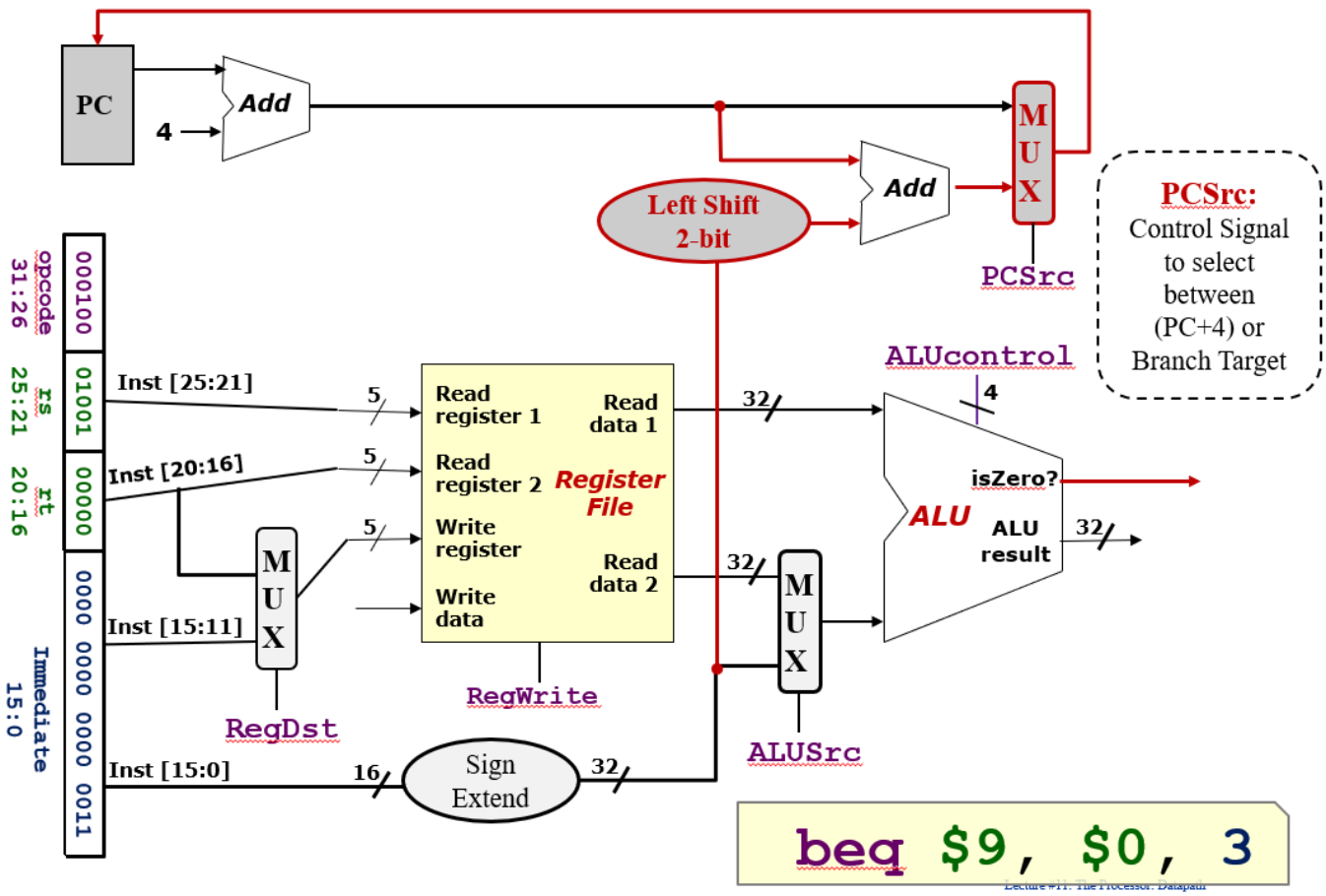
Decode & ALU Stage: Non-Branch Instructions



ALU Stage: Branch Instructions

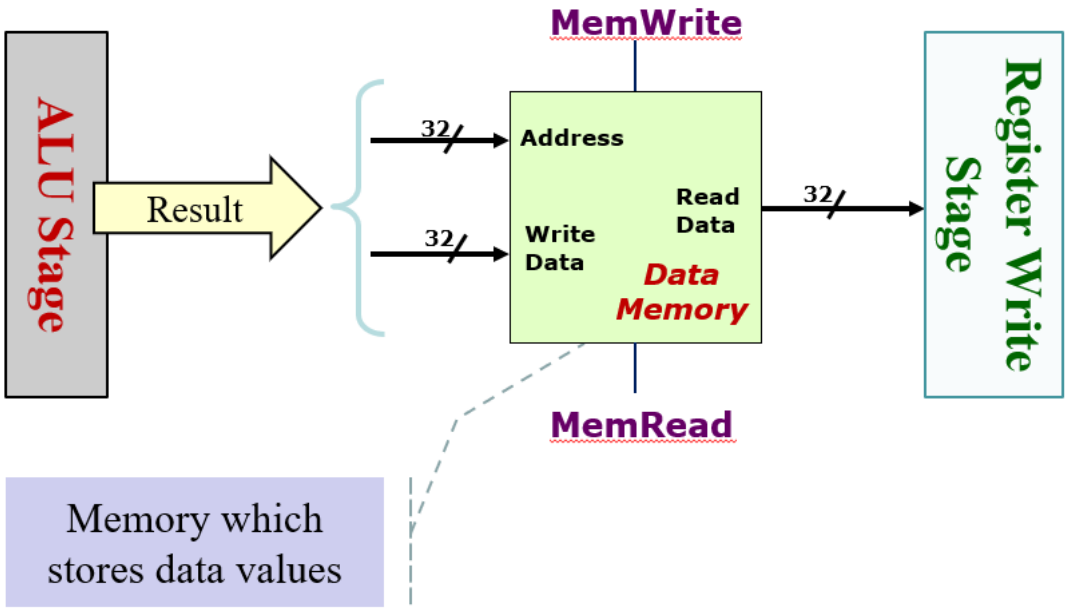
- Branch Instruction is harder as we need to perform two calculations, for example `beq $9, $0, 3`

1. Branch Outcome
    - Use ALU to compare the register
    - The 1-bit `isZero` signal is enough to handle equal/not equal-check.
2. Branch Target Address:
    - Introduce additional logic to calculate the address
    - Need PC (from Fetch Stage)
    - Need Offset (from Decode Stage)



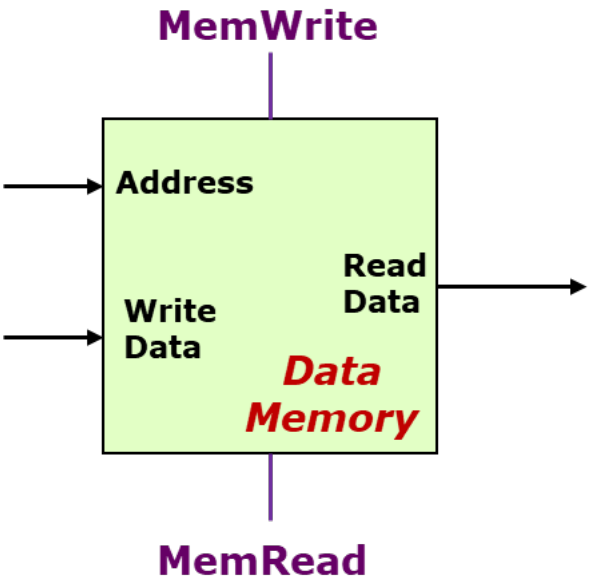
6.5.4 Memory Stage

- Instruction Memory Access Stage:
  - Only the load and store instructions need to perform operation in this stage:
    - Use memory address calculated by ALU stage
    - Read from or write to data memory
  - All other instruction remain idle
    - Result from ALU stage will pass through to be used in Register Write stage if applicable
- Input from previous stage (ALU):
  - Computation result to be used as memory address (if applicable)
- Output to next stage (Register Write):
  - Result to be stored (if applicable)



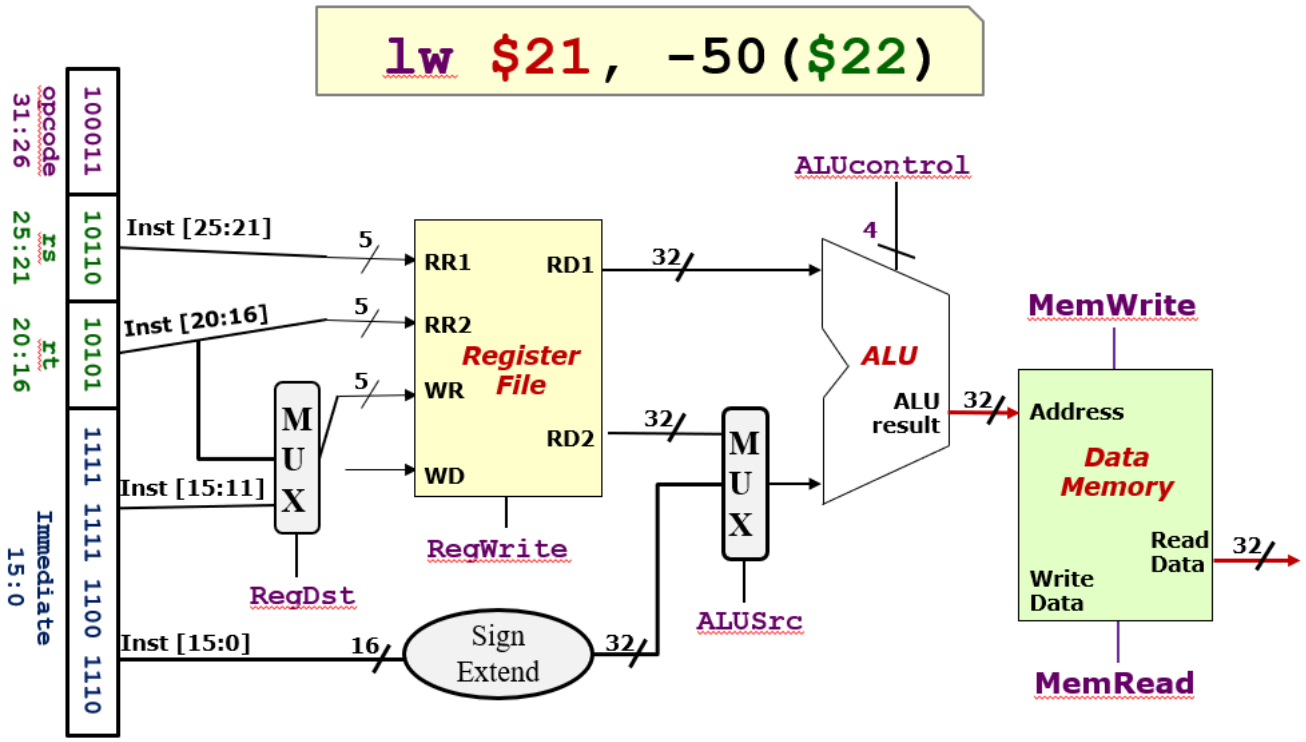
Element: Data Memory

- Storage element for the data of a program
- Inputs: (1) Memory address, (2) data to be written (Write Data) for store instructions
- Output: Data read from memory (Read Data) for load instructions



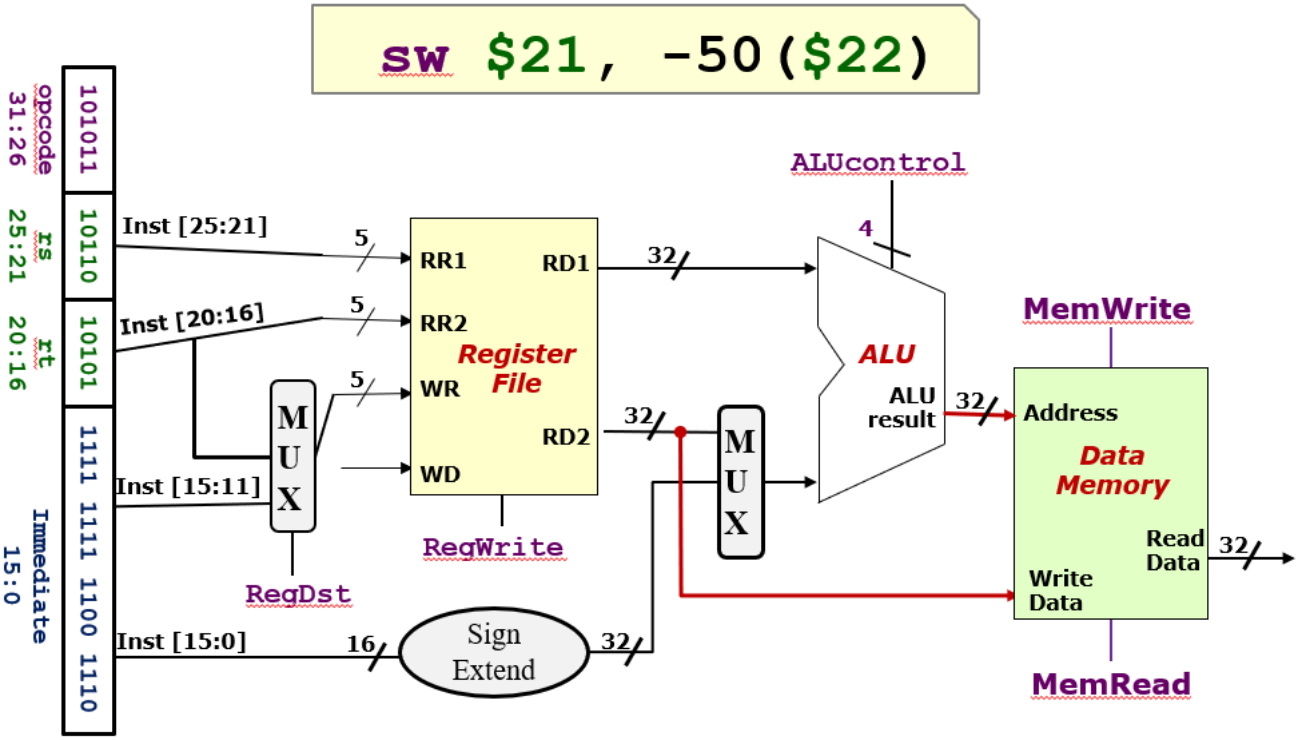
Load Instruction

- **Only relevant parts of Decode and ALU Stages are shown**



Store Instruction

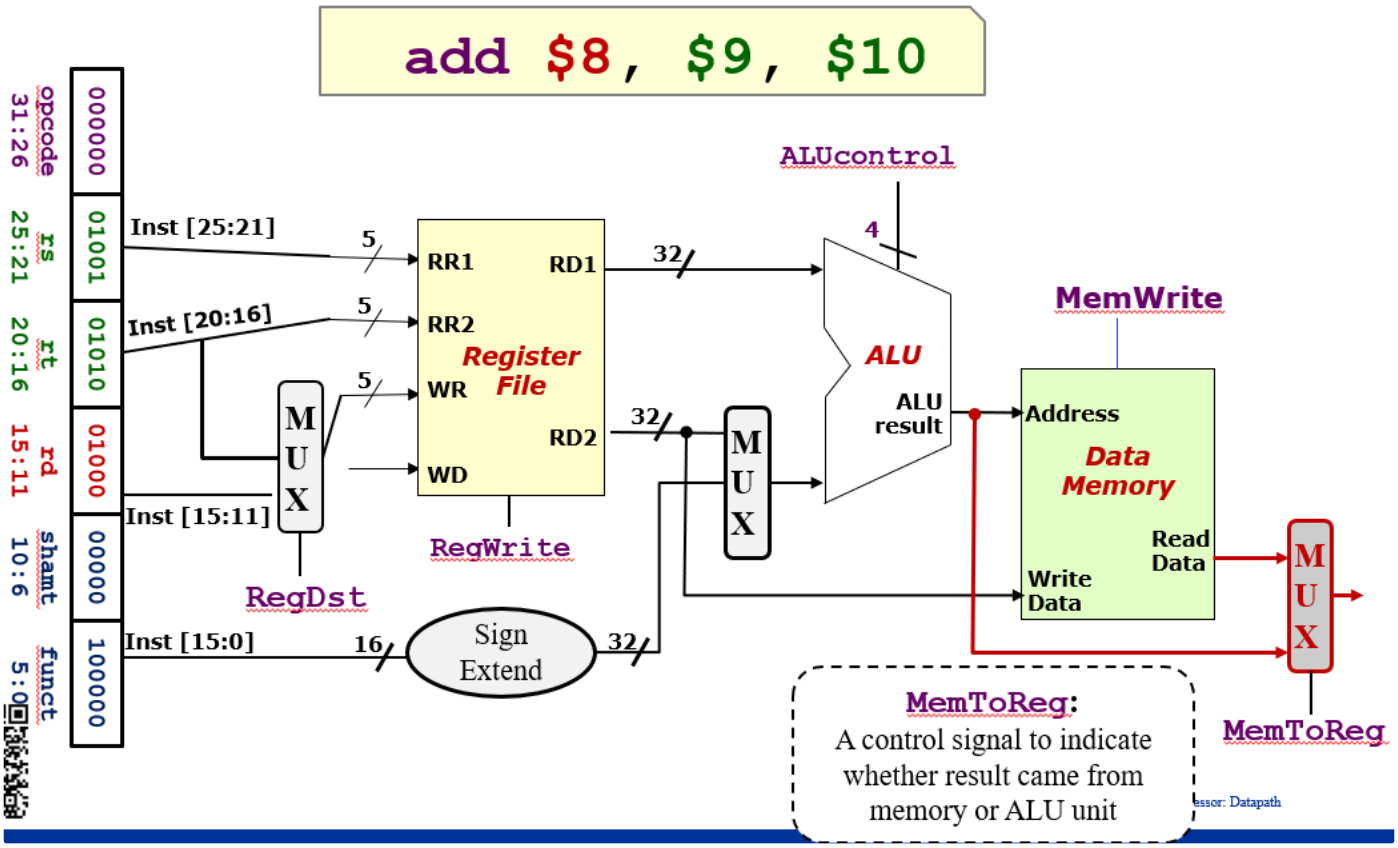
- **Need *Read Data 2* (from Decode stage) as the *Write Data***



1. **lw (Load Word) 指令**：它的功能是从内存中加载一个字（word，通常为32位）到寄存器中。该指令需要提供一个基址寄存器和一个偏移量来确定要读取的内存地址。例如，指令 `lw $t0, 4($t1)` 会从内存地址 `$t1 + 4` 加载一个字并存放到寄存器 `$t0` 中。
2. **sw (Store Word) 指令**：与 `lw` 指令相反，`sw` 的功能是将一个寄存器中的字存储到内存中。同样，这需要一个基址寄存器和一个偏移量来确定要写入的内存地址。例如，指令 `sw $t0, 4($t1)` 会将寄存器 `$t0` 中的内容存储到内存地址 `$t1 + 4` 。

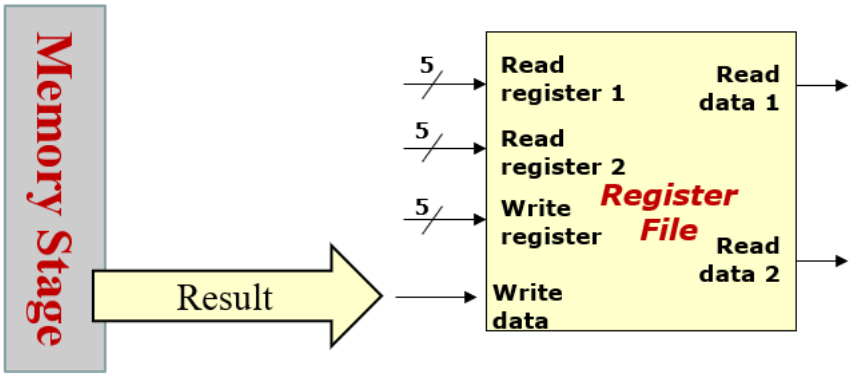
For Non-Memory Instruction (`add`)

- **Add a multiplexer to choose the result to be stored**



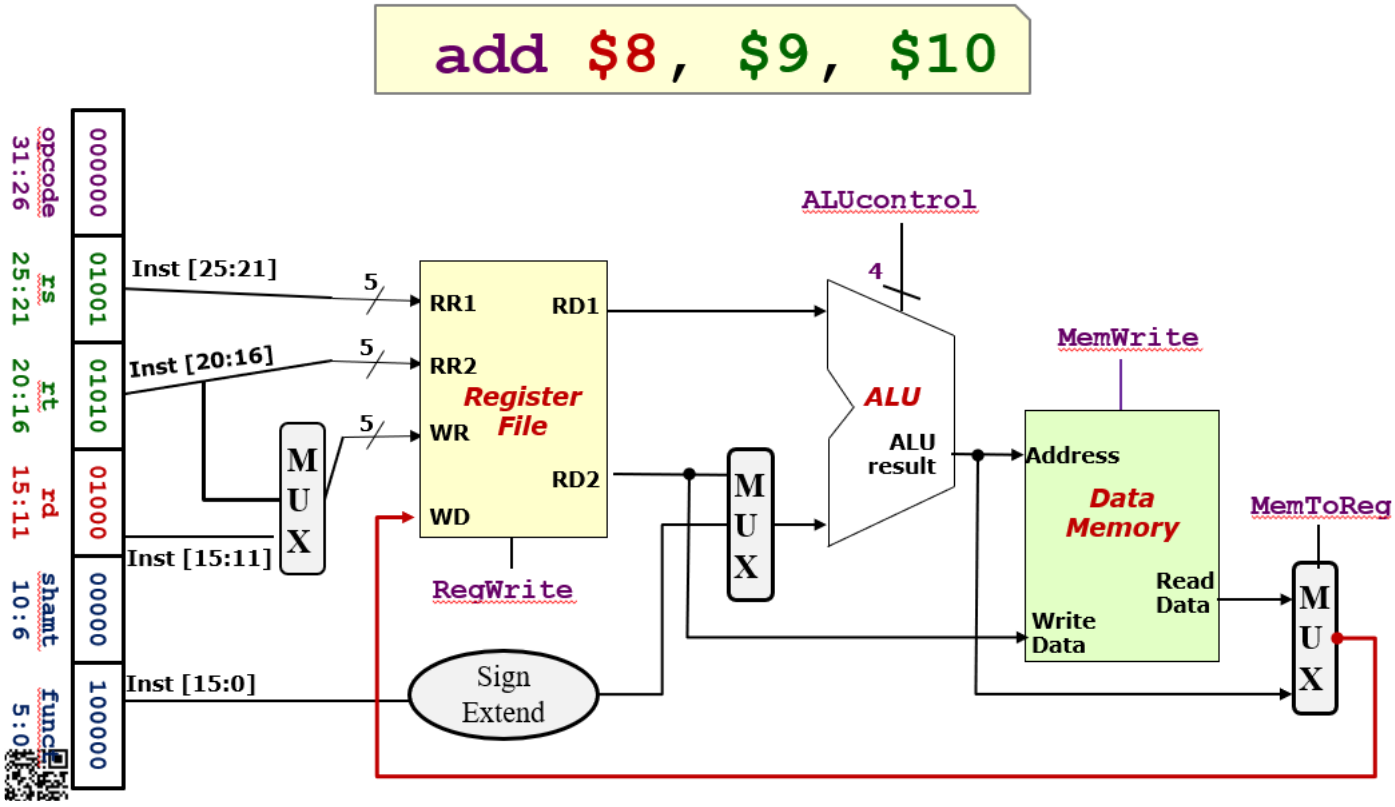
6.5.5 Register Write Stage

- Instruction Register Write Stage:
  - Most instructions write the result of some computation into a register
    - Examples: arithmetic, logical, shifts, loads, set-less-than
    - Need destination register number and computation result
  - Exceptions are stores, branches, jumps
    - These are no result to be written
    - These instructions remain idle in this stage
- Input from previous stage (Memory):
  - Computation result either from memory or ALU

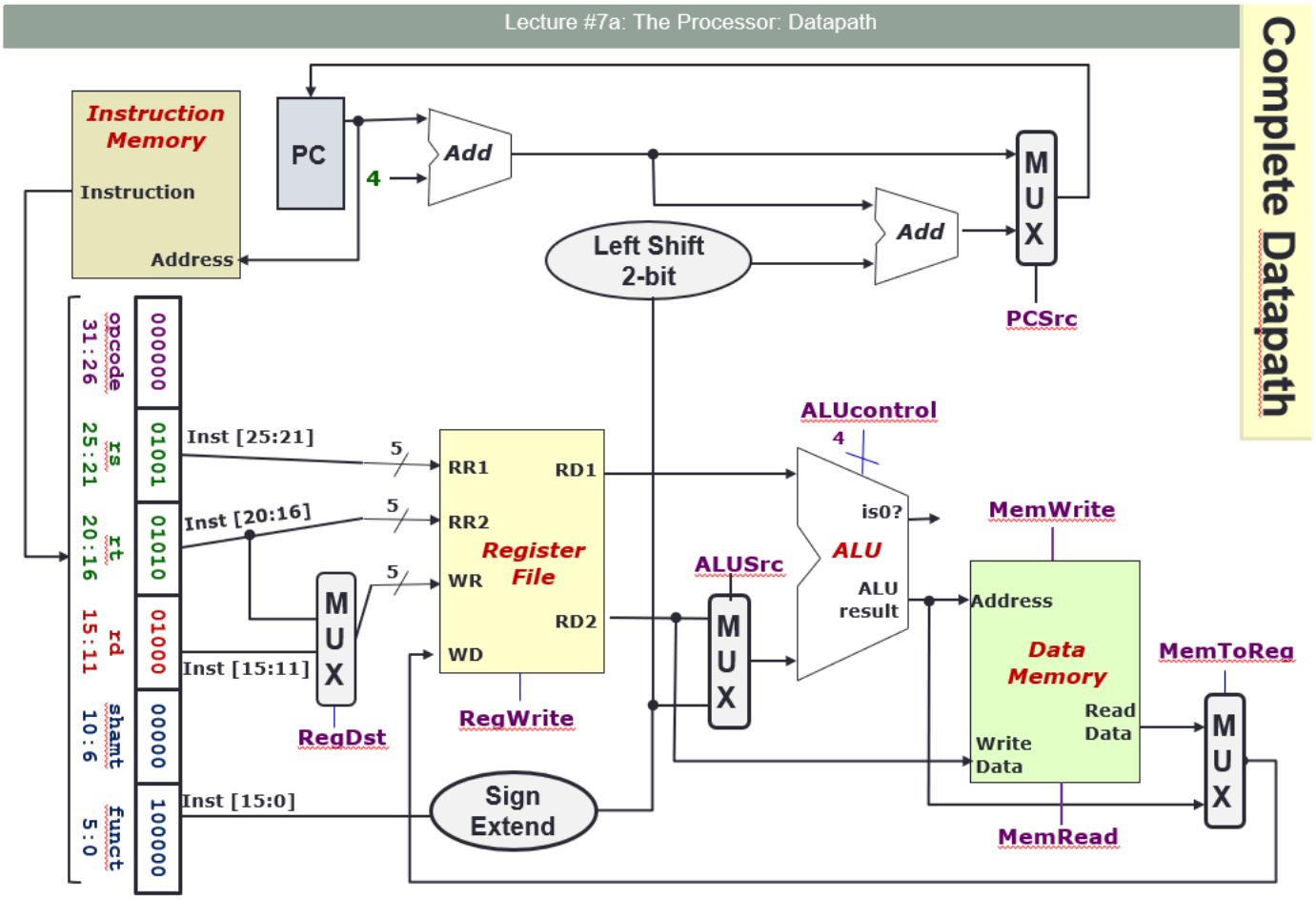


- Result Write stage has no additional element:
  - Basically just route the correct result into register file
  - The Write Register number is generated way back in the Decode Stage

Routing

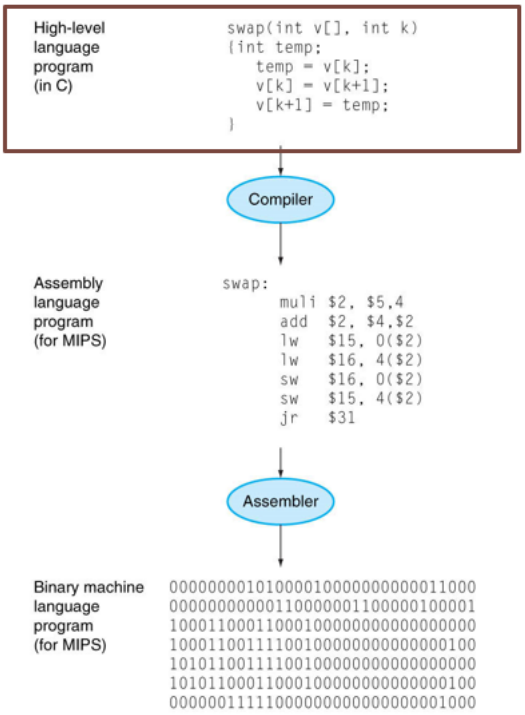


6.6 Complete Datapath


Lecture #7a: The Processor: Datapath
Complete Datapath
Instruction Memory
Instruction
Address
PC
4
Add
Add
Left Shift 2-bit
M U X
PCSrc
ALUcontrol
4
opcode 31:26
rs 25:21
rt 20:16
rd 15:11
shamt 10:6
funct 5:0
000000
01001
01010
01000
00000
100000
Inst [25:21]
Inst [20:16]
Inst [15:11]
Inst [15:0]
5
5
5
RR1
RR2
WR
WD
RD1
RD2
Register File
RegWrite
RegDst
ALUSrc
is0?
ALU
ALU result
MemWrite
Address
Data Memory
Write Data
Read Data
MemRead
MemToReg
Sign Extend

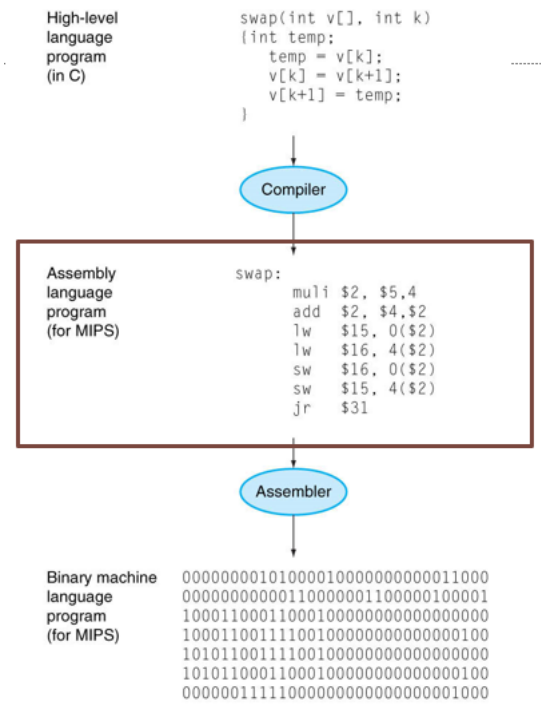
7 - The Processor: Datapath

7.1 Brief Recap



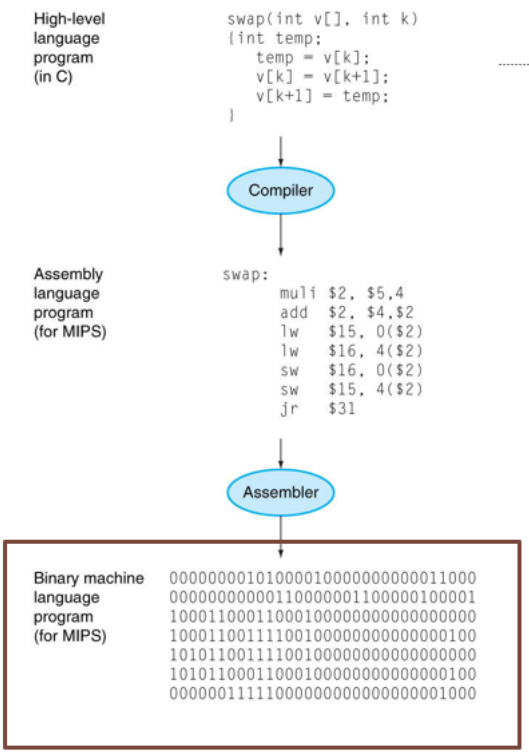
- Write program in high-level language (e.g., **C**)

```
if(x != 0) {
  a[0] = a[1] + x;
}
```



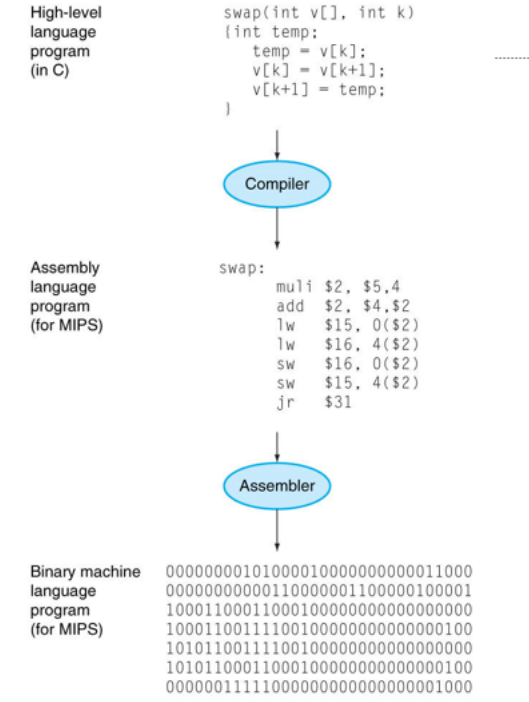
- Compiler translates to assembly language (e.g., **MIPS**)

```
      beq $16, $0, Else
      lw  $8, 4($17)
      add $8, $8, $16
      sw  $8, 0($17)
Else:
```

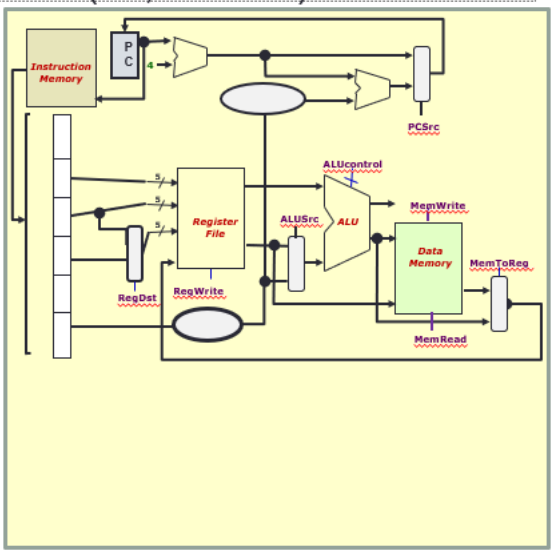


- Assembler translates to machine code (i.e., **binaries**)

```
0001 0010 0000 0000
0000 0000 0000 0011

1000 1110 0010 1000
0000 0000 0000 0100

0000 0010 0000 1000
0100 0000 0001 0100

1010 1110 0010 1000
0000 0000 0000 0000
```



- Processor executes the machine code (i.e., **binaries**)



7.2 From C to Execution

- We play the role of Programmer, Compiler, Assembler, and Processor
  - Program:

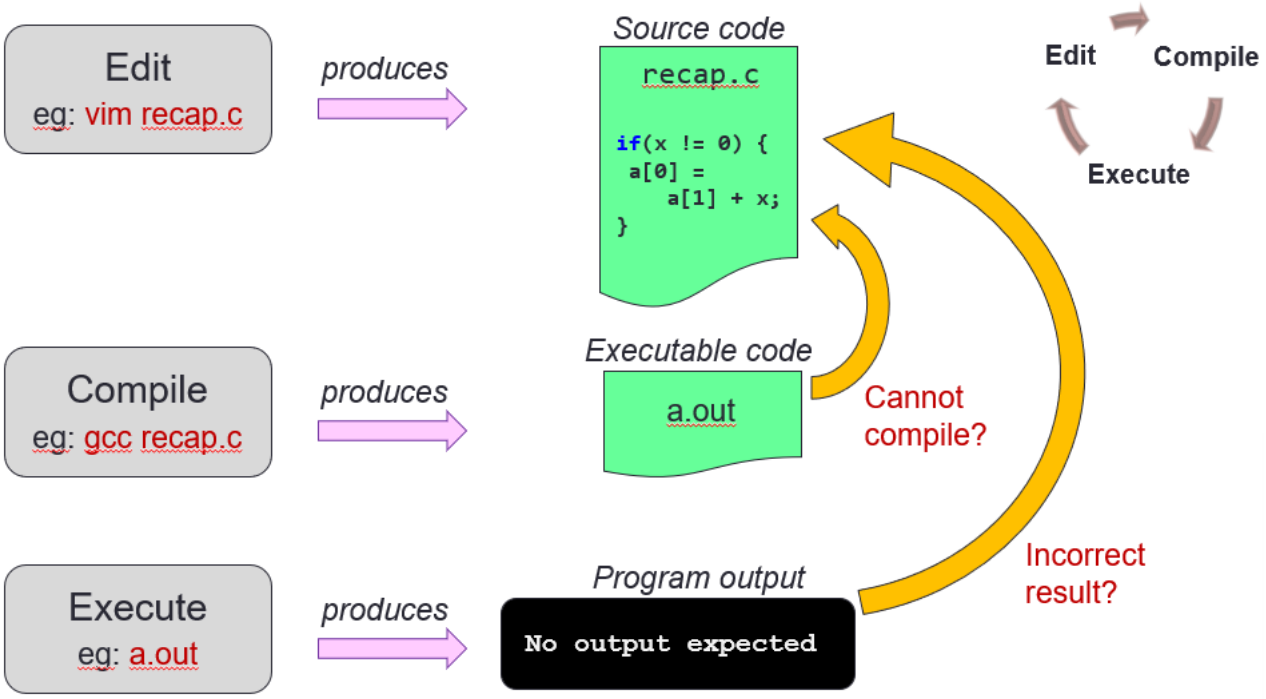
```c
if (x != 0) {
    a[0] = a[1] + x;
}
```

  - Programmer: Show the workflow of compiling, assembling, and executing C program

- Compiler: Show how the program is compiled into MIPS
- Assembler: Show how the MIPS is translated into binaries
- Processor: Show how the datapath is activated in the processor

7.2.1 Writing C Program

- Edit, Compile, Execute: Lecture #2, Slide 5



7.2.2 Compiling to MIPS

Key Idea

- Key Idea #1:

  Compilation is a structed process

```
1 if (x ≠ 0) {
2     a[0] = a[1] + x;
3 }
```

  Each structure can be compiled independently

Inner Structure

```
a[0] = a[1] + x;
```

Outer Structure

```
if(x != 0) {


}
```

- Key Idea #2:

  Variable-to-Register Mapping

  Let the mapping be:

Variable	Register Name	Register Number
x	$s0	$16
a	$s1	$17

Common Technique

- Common Technique #1:

  Invert the condition for shorter code

Outer Structure

```
if(x != 0) {



}
```

Outer MIPS Code

```
        beq $16, $0, Else

        # Inner Structure

Else:
```

- Common Technique #2:

  Break complex operations, use temp register

Inner Structure

```
a[0] = a[1] + x;
```

Simplified Inner Structure

```
$t1  = a[1];
$t1  = $t1 + x;
a[0] = $t1;
```

- Common Technique #3:

  Array access is `lw`, array update is `sw`

Simplified Inner Structure

```
$t1  = a[1];
$t1  = $t1 + x;
a[0] = $t1;
```

Inner MIPS Code

```
lw  $8, 4($17)
add $8, $8, $16
sw  $8, 0($17)
```

Common Error

- Common Error #1:

  Assume that the address of the next word can be found by incrementing the address in a register by 1 instead of by the word size in bytes

  Example:

  `$t1 = a[1]`

  is translated to:

  `lw $8, 4($17)`

  instead of

  `lw $s8, 1($17)`

Finalize

- Last Step:

  Combine the two structures logically

Inner MIPS Code

```
lw  $8, 4($17)
add $8, $8, $16
sw  $8, 0($17)
```

Outer MIPS Code

```
      beq $16, $0, Else

      # Inner Structure

Else:
```

Combined MIPS Code

```
      beq $16, $0, Else
      lw  $8, 4($17)
      add $8, $8, $16
      sw  $8, 0($17)
Else:
```

7.2.3 Assembling to Binaries

- Instruction Types Used:

1. R-Format: `opcode $rd, $rs, $rt`

2. I-Format: `opcode $rt, $rs, immediate`

3. Branch:
   - Use I-format
   - PC = (PC+4) + (`immediate` x 4)
4. `beq $16, $0, Else`
   - Compute immediate value
     - `immediate` = 3
   - Fill in fields

6	5	5	16
4	16	0	3

   - Convert to binary

6	5	5	16
4	16	0	3

```
      beq $16, $0, Else
      lw  $8, 4($17)
      add $8, $8, $16      +3
      sw  $8, 0($17)
Else:
```

5. `lw $8, 4($17)`
   - Filled in fields (Refer to MIPS Reference data)

6	5	5	16
35	17	8	4

   - Convert to binary

100011	10001	01000	0000000000000100

```
0001 0010 0000 0000 0000 0000 0000 0011
      lw  $8, 4($17)
      add $8, $8, $16
      sw  $8, 0($17)
Else:
```

6. `add $8, $8, $16`
   - Filled in fields

6	5	5	5	5	6
0	8	16	8	0	32

   - Convert to binary

000000	01000	10000	01000	00000	100000

```
0001 0010 0000 0000 0000 0000 0000 0011
1000 1110 0010 1000 0000 0000 0000 0100
      add $8, $8, $16
      sw  $8, 0($17)
Else:
```

7. `sw $8, 0($17)`
   - Filled in fields

6	5	5	16
43	17	8	0

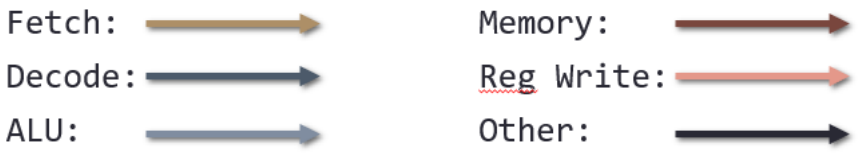
   - Convert to binary

101011	10001	01000	0000000000000000

```
0001 0010 0000 0000 0000 0000 0000 0011
1000 1110 0010 1000 0000 0000 0000 0100
0000 0001 0001 0000 0100 0000 0010 0000
      sw  $8, 0($17)
Else:
```

7.2.4 Execution (Datapath)

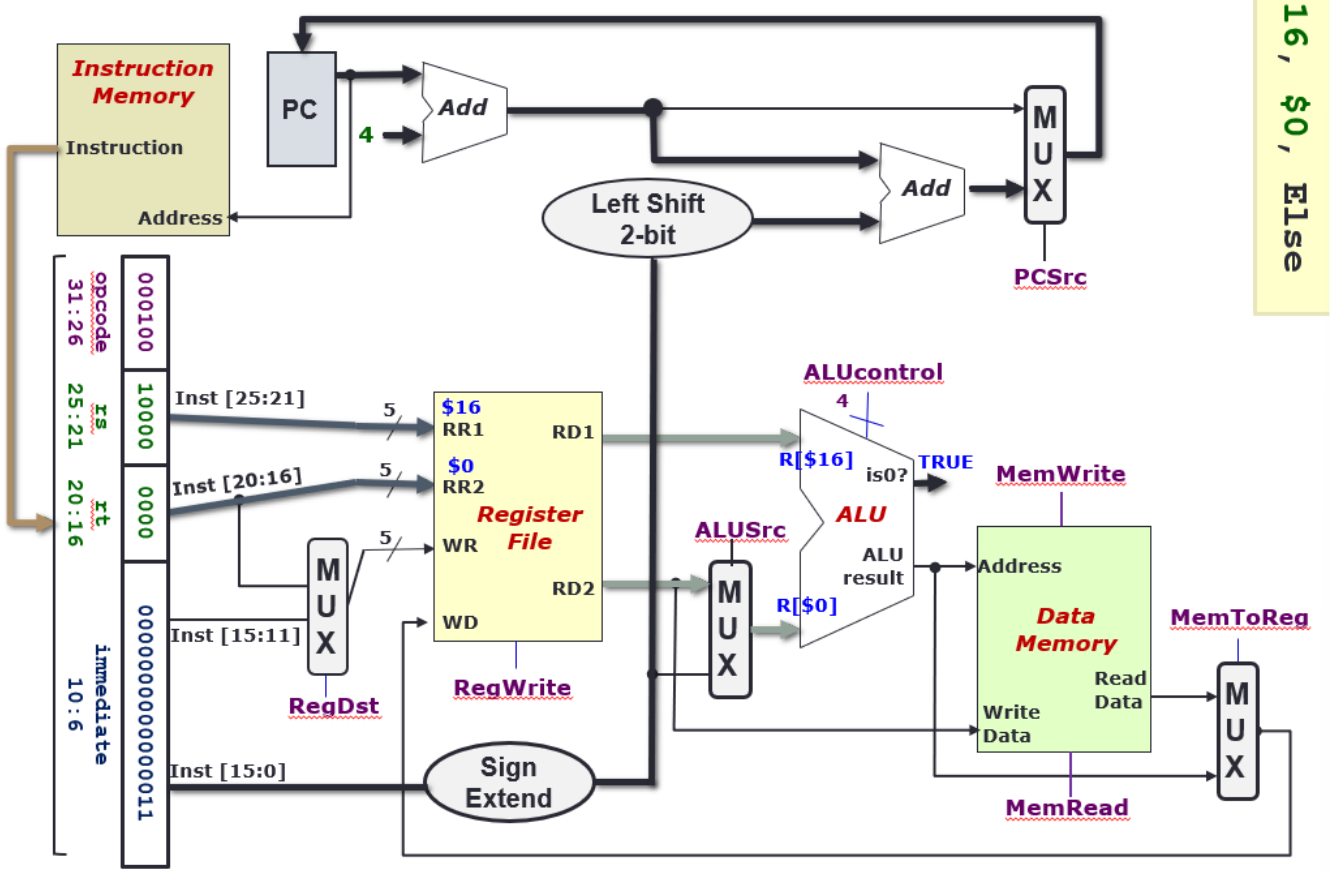
- Given the binary
  - Assume two possible executions
    1. `$16 == $0` (shorter)
    2. `$16 ≠ $0` (larger)
  - Convention:



Lecture #7a: The Processor: Datapath

beq $16, $0, Else

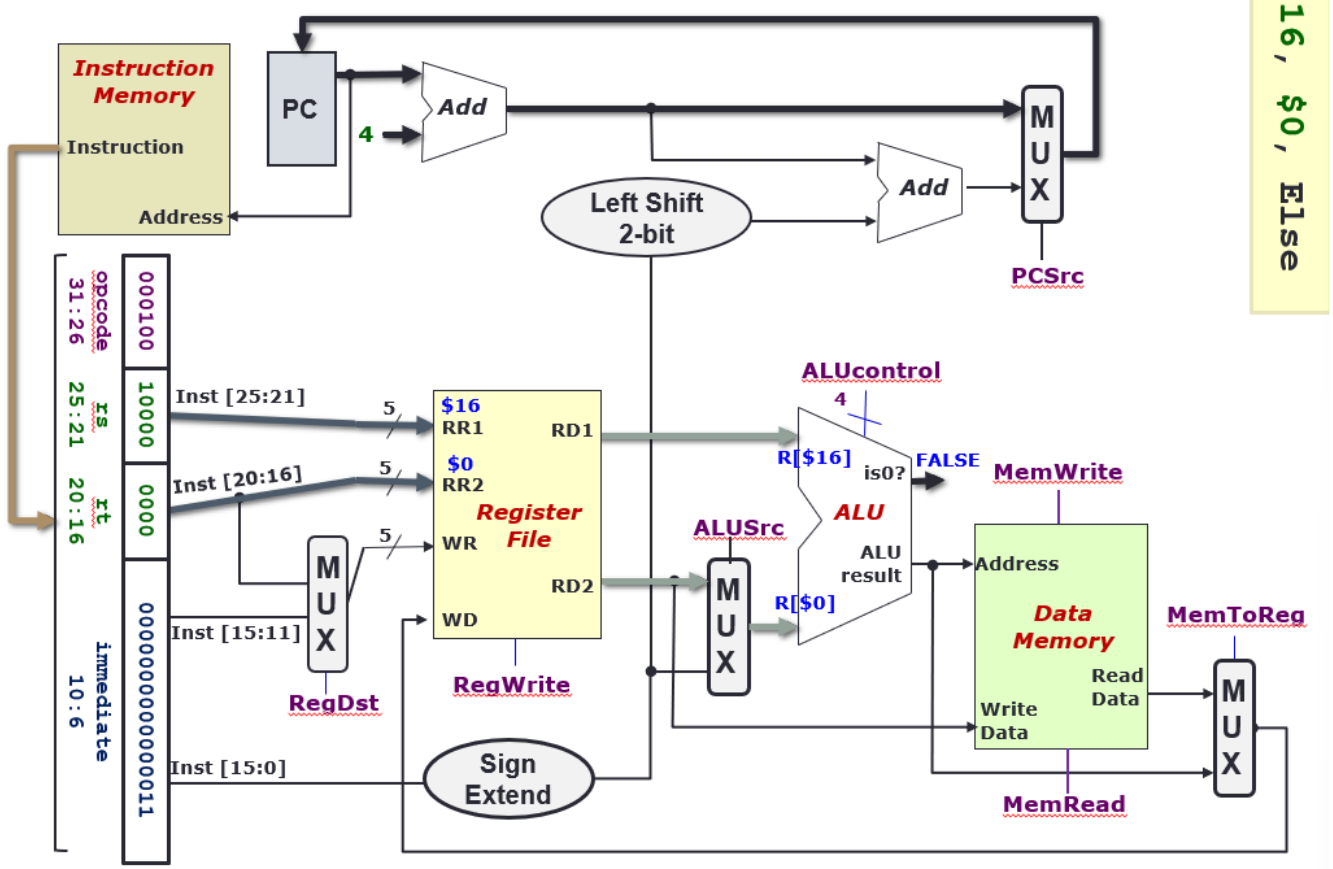
- Assume $16 == $0



Lecture #7a: The Processor: Datapath

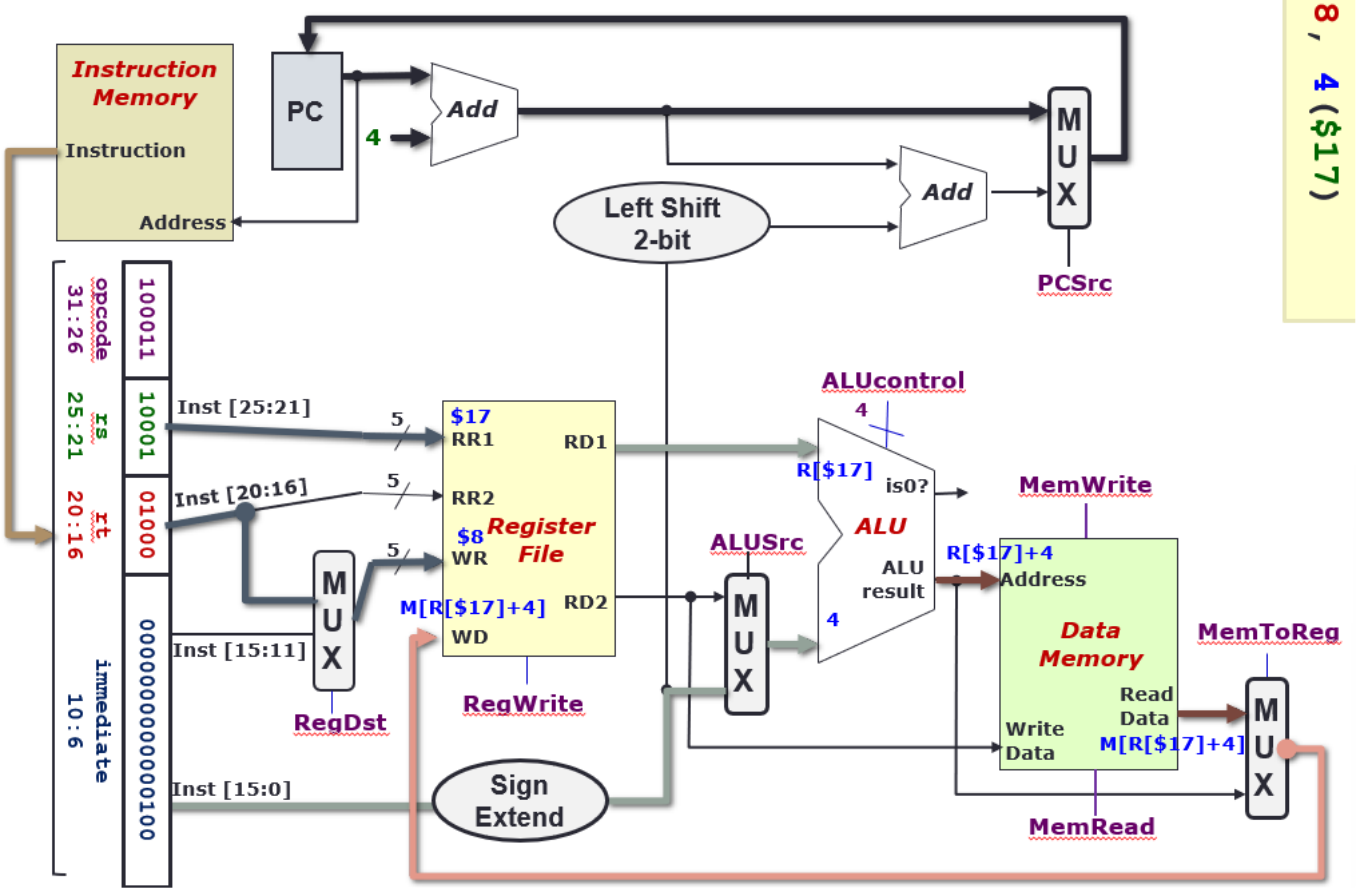
beq $16, $0, Else

- Assume $16 != $0



lw $8, 4($17)

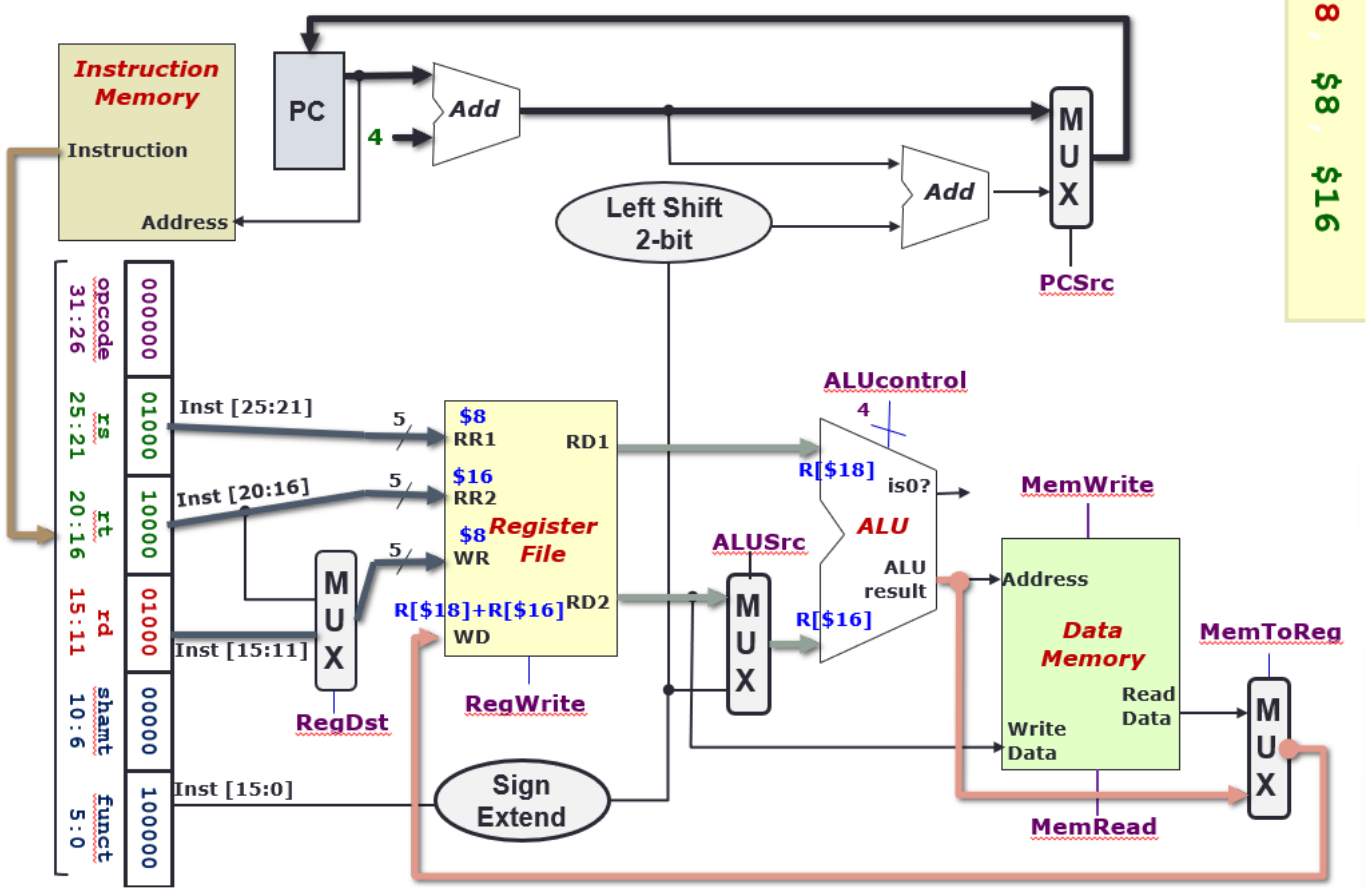
- Assume $16 != $0



Lecture #7a: The Processor: Datapath

add $8, $8, $16

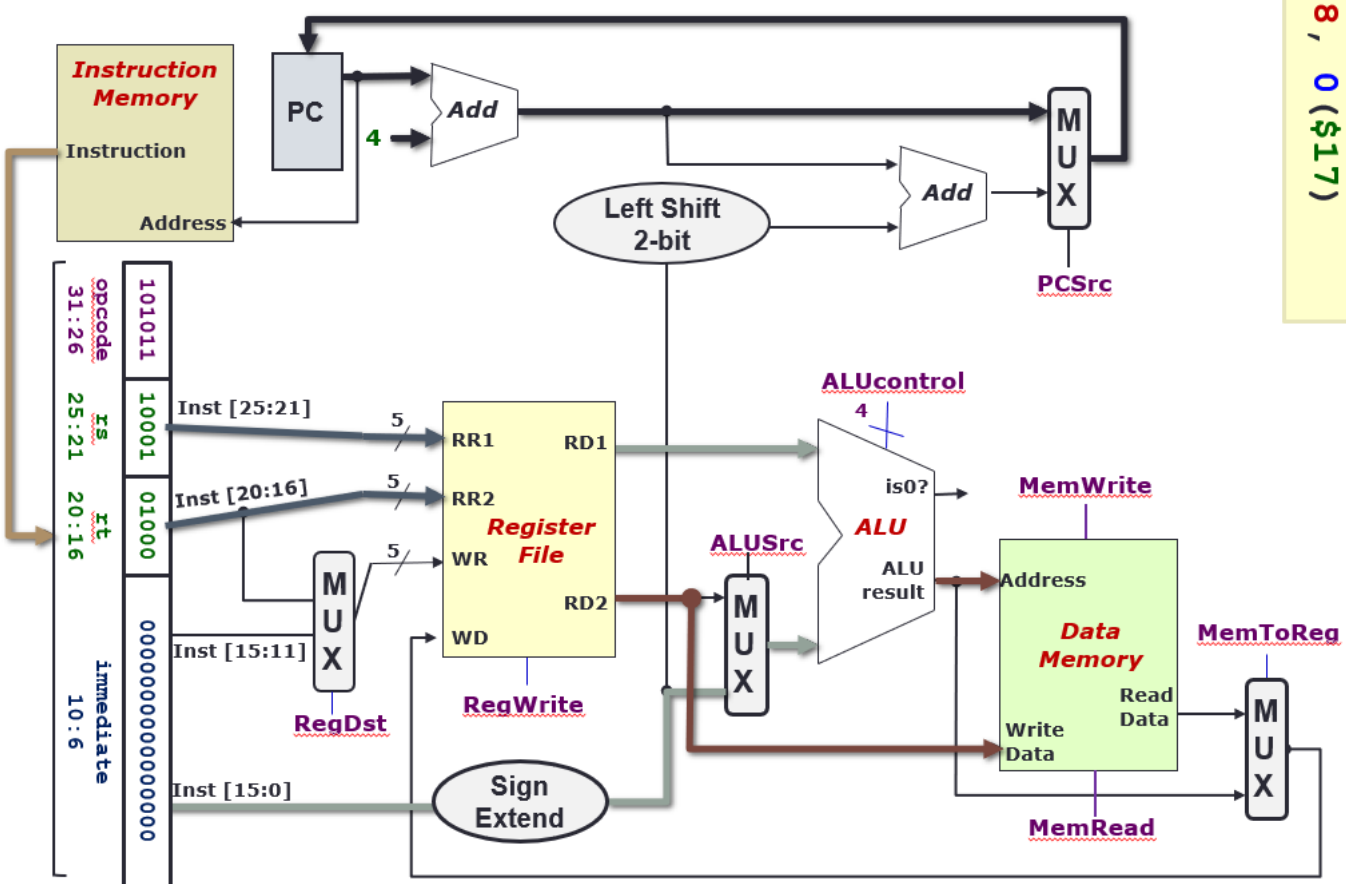
- Assume $16 != $0



Lecture #7a: The Processor: Datapath

sw $8, 0($17)

- Assume $16 != $0



8 - Processor: Control

8.1 Identified Control Signals

Control Signal	Execution Stage	Purpose
RegDst	Decode/Operand Fetch	Select the destination register number
RegWrite	Decode/Operand Fetch RegWrite	Enable writing of register
ALUSrc	ALU	Select the 2nd operand for ALU
ALUControl	ALU	Select the operation to be performed
MemRead / **MemWrite**	Memory	Enable reading/writing of data memory
MemToReg	RegWrite	Select the result to be written back to register file
PCSrc	Memory/RegWrite	Select the next PC value

8.2 Generating Control Signals: Idea

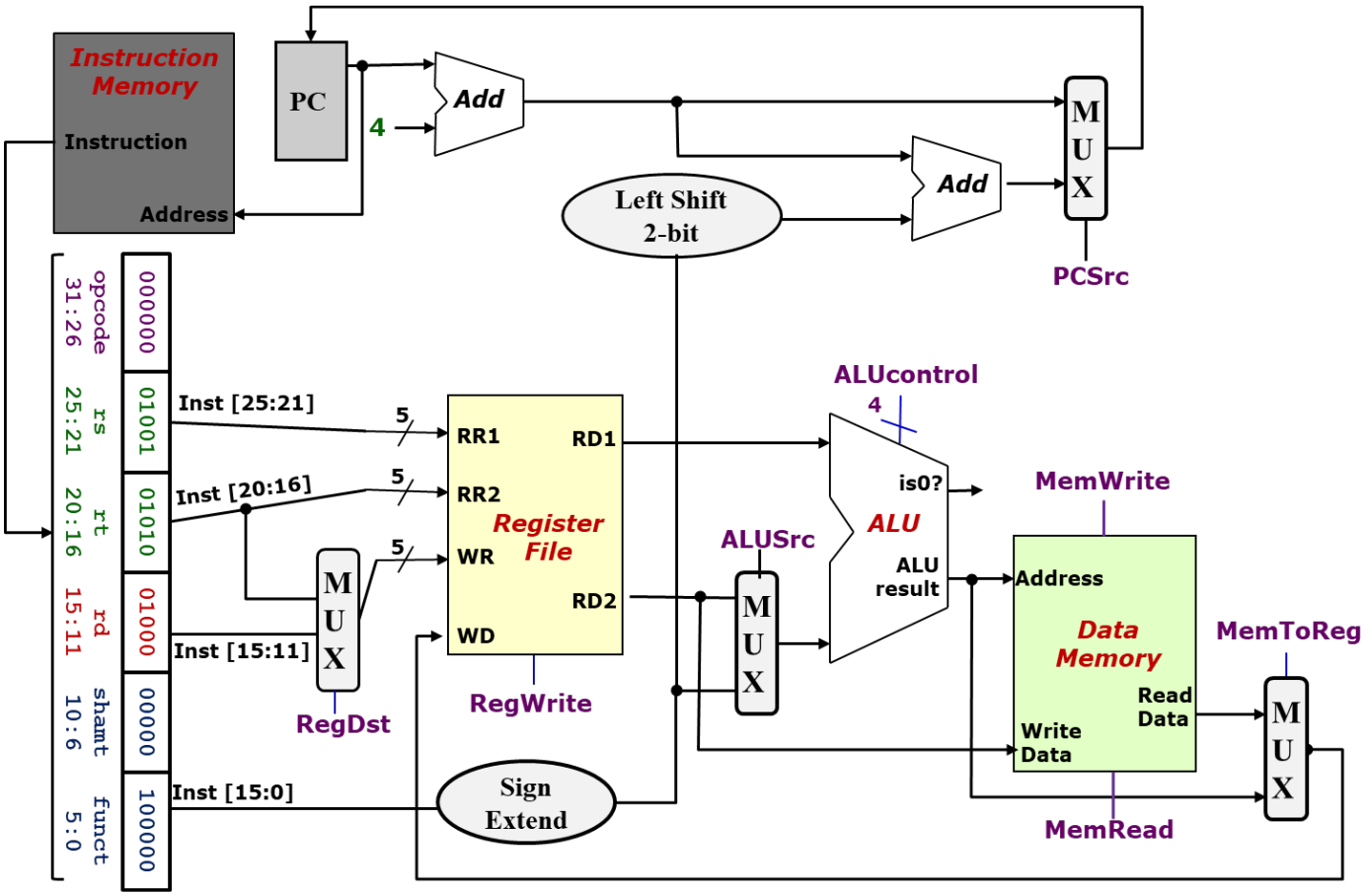
- The control signals are generated based on the instruction to be executed:
  - `opcode` → Instruction Format
  - Example:
    - R-Format instruction → `RegDst` = 1 (use `Inst[15:11]`)
  - R-Type instruction has additional information:
    - The 6-bit `funct` (function code, `Inst[5:0]`) field
- Idea:
  - Design a combinatorial circuit to generate these signals based on Opcode and possibly Function code
    - A control unit is needed

控制信号

1. 控制信号的生成：
   - 控制信号是基于要执行的指令而生成的。这些信号告诉数据路径硬件如何执行指令。例如应该执行哪种算术或逻辑操作，数据应该来自哪里已经结果应该存储在哪里

2. opcode
    - 所有MIPS指令的开始部分都有一个操作码(opcode)，它决定了指令的基本操作类型。通过解码(decode)这个操作码，可以知道要执行的指令类型，从而生成相应的控制信号
3. 指令格式与 RegDst
    - 例如对于R-Format（寄存器格式）的指令，有一个控制信号 RegDst 决定目标寄存器的选择。如果在R-Format指令中 RegDst 设置为1，则意味着目标寄存器的信息来自于 Inst[15:11] 字段
4. **R-Type指令的额外信息**：
    - R-Type指令除了操作码外，还有一个6位的函数代码（funct）字段，即 Inst[5:0] 。这个函数代码进一步指定了R-Type指令的具体操作，例如加法、减法等。
5. **主要思想**：
    - 设计一个组合逻辑电路，根据指令的操作码（Opcode）和可能的函数代码（Function code）生成这些控制信号。
    - 为了生成和管理这些控制信号，需要一个控制单元。

8.3 The Control Units

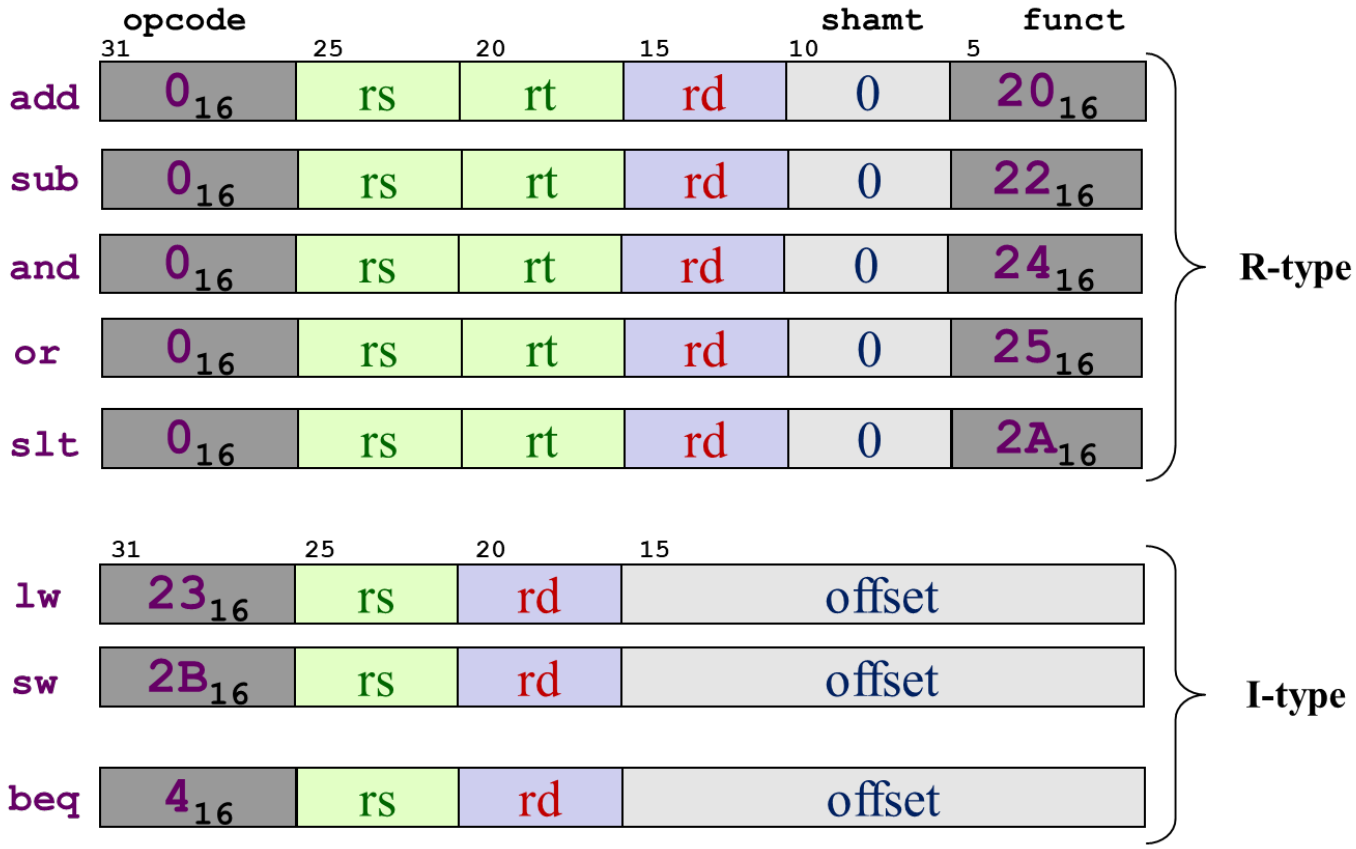


8.3.1 Implement the Control Unit

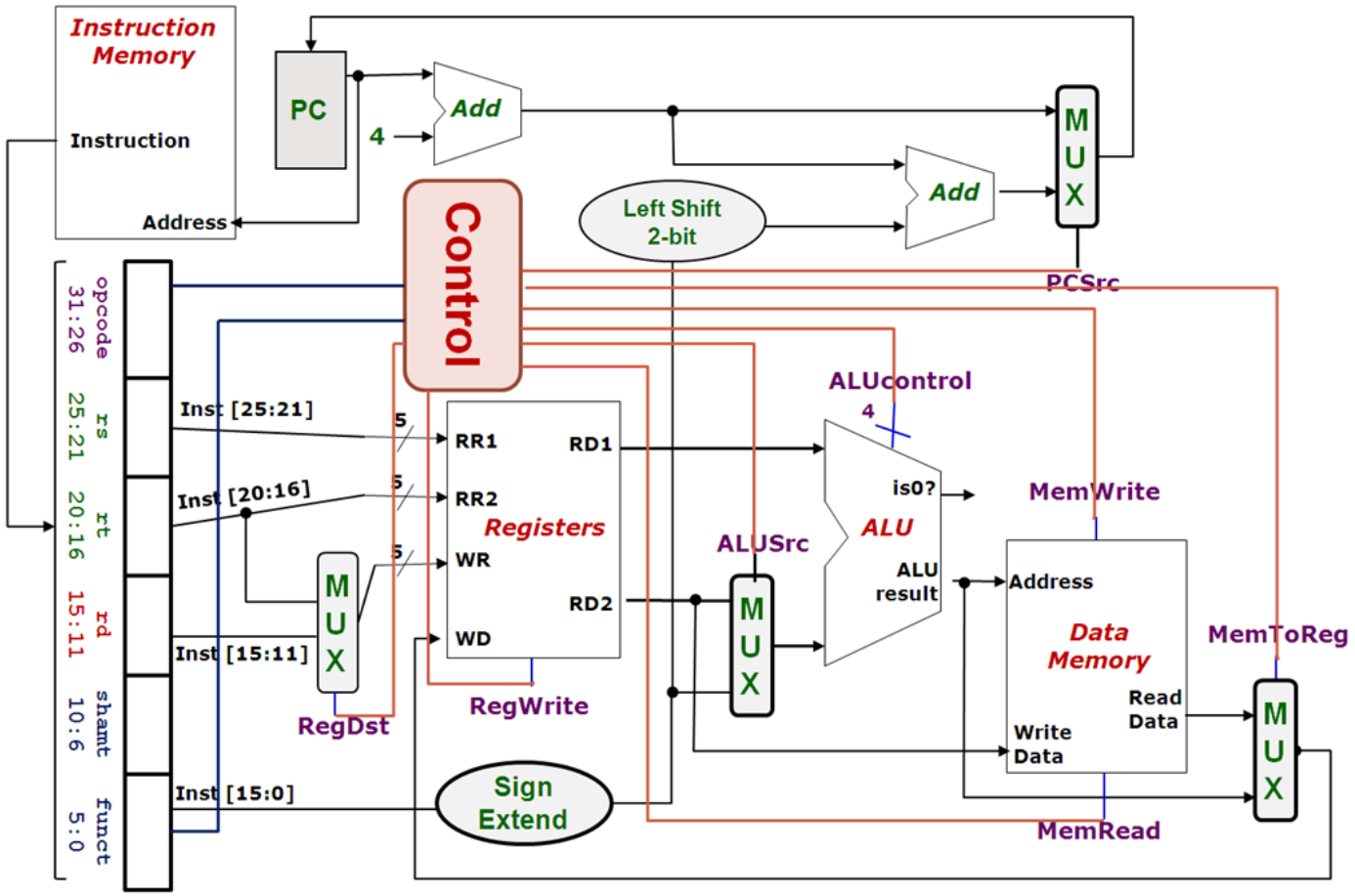
- Approach:
    - Take note of the instruction subset to be implemented:
        - `opcode` and function code

- Go through each signal:
  - Observe how the signal is generated based on the instruction opcode and/or function code
- Construct truth table
- Design the control unit using logic gates

8.3.2 MIPS Instruction Subset



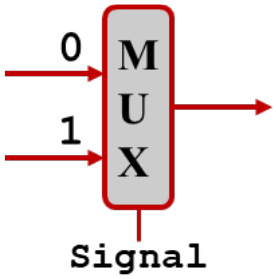
8.4 Control Signals



8.4.1 RegDst

> 这个控制信号决定了要写入的目标寄存器

- False (0): Write register = `Inst[20:16]`
- True (1): Write register = `Inst[15:11]`



8.4.2 RegWrite

> 决定是否将新的值写入寄存器

- False (0): No register write
- True (1): New value will be written

8.4.3 ALUSrc

> 这个信号决定了ALU（算术逻辑单元）的第二个操作数来源

- False (0): Operand2 = Register Read Data 2
- True (1): Operand2 = SignExt(Inst[15:0])

8.4.4 MemRead

> 决定是否执行内存读取操作

- False (0): Not performing memory read access
- True (1): Read memory using `address`

8.4.5 MemWrite

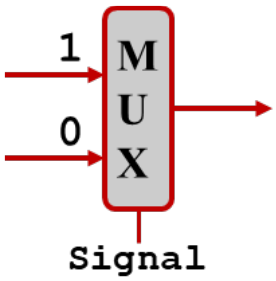
> 决定是否执行内存写入操作

- False (0): Not performing memory write operation
- True (1): memory[address] ← Register Read Data 2

8.4.6 MemToReg

> 这个信号决定了要写入寄存器的数据来源

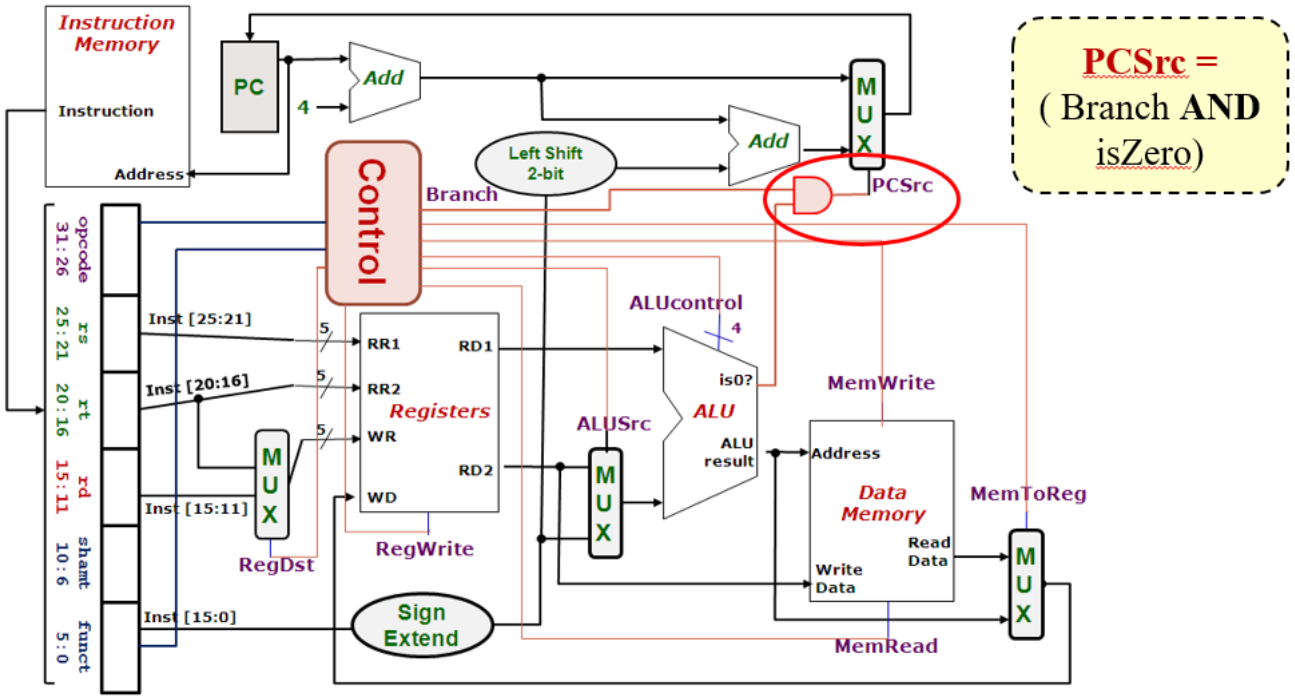
- True (1): Register write data = Memory read data
- False (0): Register write data = ALU result
- Important: The input of MUX is swapped in this case



8.4.7 PCSrc

> 这个控制信号基于ALU的 `isZero` 信号来确定分支指令的实际结果（是否执行分支）

- The `isZero` signal from the ALU gives us the actual branch outcome (taken/not taken)
- Idea: “If instruction is a branch AND taken, then…”
- False (0): Next PC = PC + 4
- True (0): Next PC = SignExt(Inst[15:0]) << 2 + (PC + 4)



Summary

Observation so far:

- The signals discussed so far can be generated by `opcode` directly
  - Function code is not needed up to this point
- A major part of the controller can be built based on `opcode` alone

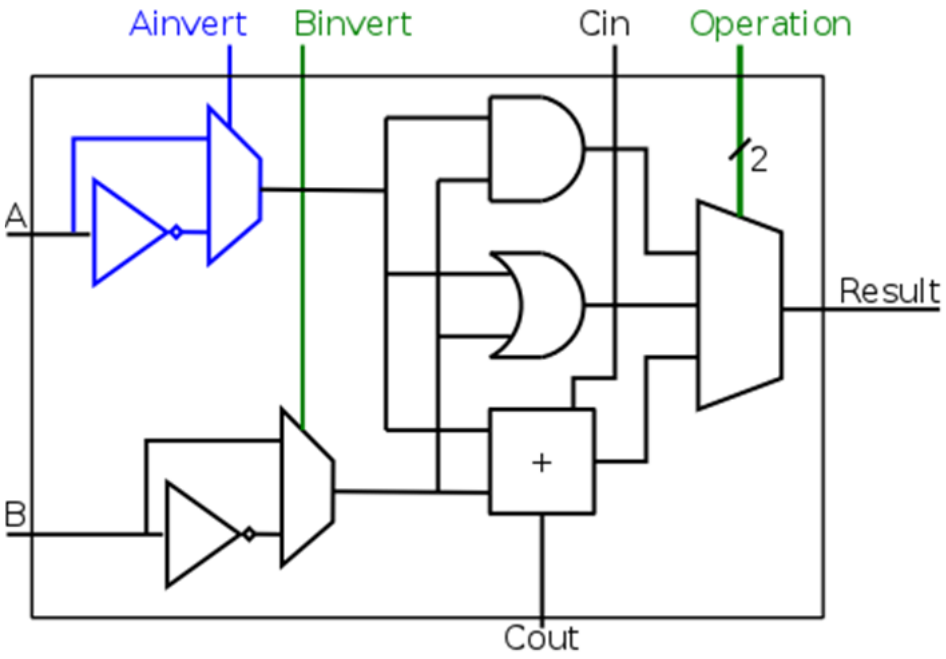
8.5 ALU Control Signal

- The ALU is a combinatorial circuit:
  - Capable of performing several arithmetic operations

ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR

8.5.1 One Bit At A Time

- A simplified 1-bit MIPS ALU can be implemented as follows:



- 4 control bits are needed:
  - `Ainvert`
    - 1 to invert input A
  - `Binvert`
    - 1 to invert input B
  - `Operation` (2-bit)
    - To select one of the 3 results

ALUcontrol			Function
Ainvert	Binvert	Operation	
0	0	00	AND
0	0	01	OR
0	0	10	add
0	1	10	subtract
0	1	11	slt
1	1	00	NOR

ALU（算术逻辑单元）通常执行多种算术和逻辑操作，而这些操作是通过内部的控制信号激活的。在某些设计中，这些控制信号中的一部分可能包括Ainvert、Binvert和Operation。

1. **Ainvert**：此信号用于控制是否应该对输入A进行求反（即位反转或数值取反）。当Ainvert设置为1时，ALU会对A的所有位进行反转（例如，从二进制的"0"变为"1"，反之亦然）。这在某些操作（例如减法）中是有用的，因为它们可以通过使用加法器和求反逻辑来简化。
2. **Binvert**：与Ainvert类似，Binvert控制是否对输入B进行反转。这也是实现减法等操作的常用技巧，因为通过求反和加法，可以很容易地在已有的硬件上执行减法。
3. **Operation**：这是一个2位的字段，它直接定义了ALU应执行的操作类型。由于它是一个2位信号，因此它可以表示4种不同的操作（例如，00表示加法，01表示减法，10表示AND操作，11表示OR操作等）。实际的操作和编码会根据具体的ALU设计而变化。

现在，让我们看看这些信号是如何协同工作来控制ALU的：

- **实现减法**：要使用ALU执行减法，我们可以利用加法器硬件来执行该操作。理论上，A - B 可以重写为 A + (-B)。因此，我们可以设置Ainvert为0（保持A不变），Binvert为1（求B的二进制反码），然后通过Operation信号告诉ALU执行加法操作。通常，还需要在B的反码上加1（即取补码），以完成从正数到负数的转换。
- **实现逻辑操作**：对于逻辑操作（如AND、OR、NOR等），Ainvert和Binvert通常会设置为0，这样A和B就保持不变。相应的操作是通过Operation字段的2位代码来指定的，这会直接控制ALU内部执行哪种逻辑操作。

这些信号的组合允许ALU利用较少的硬件资源（主要是加法器和逻辑单元）来执行一系列的算术和逻辑操作。通过巧妙地利用位反转和选择不同的操作类型，ALU可以用相对简单的方式实现复杂的功能。

8.5.2 Multilevel Decoding

- Now we can start to design for `ALUcontrol` signal, which depends on:
  - `opcode` (6-bit) field and `Function Code` (6-bit) field
- Brute Force approach
  - Use `opcode` and `function code` directly, i.e. finding expressions with 12 variables
- Multilevel Decoding approach:
  - Use some of the input to reduce the cases, then generate the full output
  - Simplify the design process, reduce the size of the main controller, potentially speedup the circuit

`ALUcontrol` 信号的生成取决于指令的两个字段：`opcode`（操作码，6位）和 `Function Code`（功能码，也是6位）。这些字段确定了CPU需要执行的具体操作。

1. **蛮力方法（Brute Force approach）**：
   - 这种方法直接使用 `opcode` 和 `function code`，即通过寻找包含12个变量的表达式来生成 `ALUcontrol` 信号。这相当于直接对所有可能的输入组合进行硬编码，非常直接但可能会非常复杂，因为它需要处理所有的 `opcode` 和 `function code` 组合。
2. **多级解码方法（Multilevel Decoding approach）**：
   - 这种方法更加巧妙。它首先使用部分输入（比如只用 `opcode` 字段）来减少需要直接解码的情况数量。基于这个初步的解码，控制逻辑可以将可能的操作范围缩小到更易管理的数量。
   - 然后，系统可能会根据需要考虑 `Function Code` 来进一步确定要执行的确切操作，从而生成完整的 `ALUcontrol` 信号。这样做简化了设计过程，因为不是每个操作都需要单独编码，同时还减小了主控制器的大小。
   - 由于解码器不必同时处理所有的12个变量，这种方法还可能加快电路的速度。处理更少的变量意味着更快的逻辑运算，从而可能提高整个处理单元的响应时间。

多级解码（Multilevel Decoding）的概念是通过在多个阶段处理输入信息来减少同时处理的变量数量，简化电路设计，提高解码效率。在第一级，解码器可能只考虑一部分输入变量并做出部分决策；在随后的级别，它会逐步考虑更多的变量，逐渐缩小操作的范围。这样做的好处是可以简化每个阶段的逻辑复杂性，减少所需硬件的数量，并提高操作速度。

8.5.3 Intermediate Signal: `ALUop`

- Basic Idea:
  1. Use `opcode` to generate a 2-bit `ALUop` signal
     - Represents classification of the instructions

Instruction type	ALUop
lw / sw	00
beq	01
R-type	10

  2. Use `ALUop` signal and `function code` field to generate the 4-bit `ALUcontrol` signal

引入一个中间信号 ALUop 来简化控制信号的生成。这是多级解码策略的一个实例，其目的是减少复杂性并提高系统效率。以下是这个过程的详细解释：

基本思路：

1. **使用 opcode 生成2位的 ALUop 信号**：
    - 在这个阶段，系统读取指令的 opcode （操作码），这是指令中的一个字段，表示要执行的操作的类型（例如，加载、存储、分支、算术运算等）。然后，这个 opcode 被解码为一个更简单的2位信号 ALUop ，它表示指令的分类。不同的 opcode 将导致不同的 ALUop 信号。
    - 例如，表中列出了三种类型的指令（ lw / sw ， beq ，和R类型），每种类型都被分配了一个特定的 ALUop 代码。
2. **使用 ALUop 信号和 function code 字段生成4位的 ALUcontrol 信号**：
    - 接下来， ALUop 信号和指令中的 function code （功能码）一起用于确定确切的操作，该操作应由ALU执行。
    - function code 是指令的另一个部分，仅在某些类型的指令（如R类型）中使用，它提供了关于应执行的确切算术或逻辑操作的更多信息。
    - 根据 ALUop 和 function code 的组合，生成一个4位的 ALUcontrol 信号，该信号直接控制ALU，告诉它要执行的确切操作（例如，加、减、与、或等）。

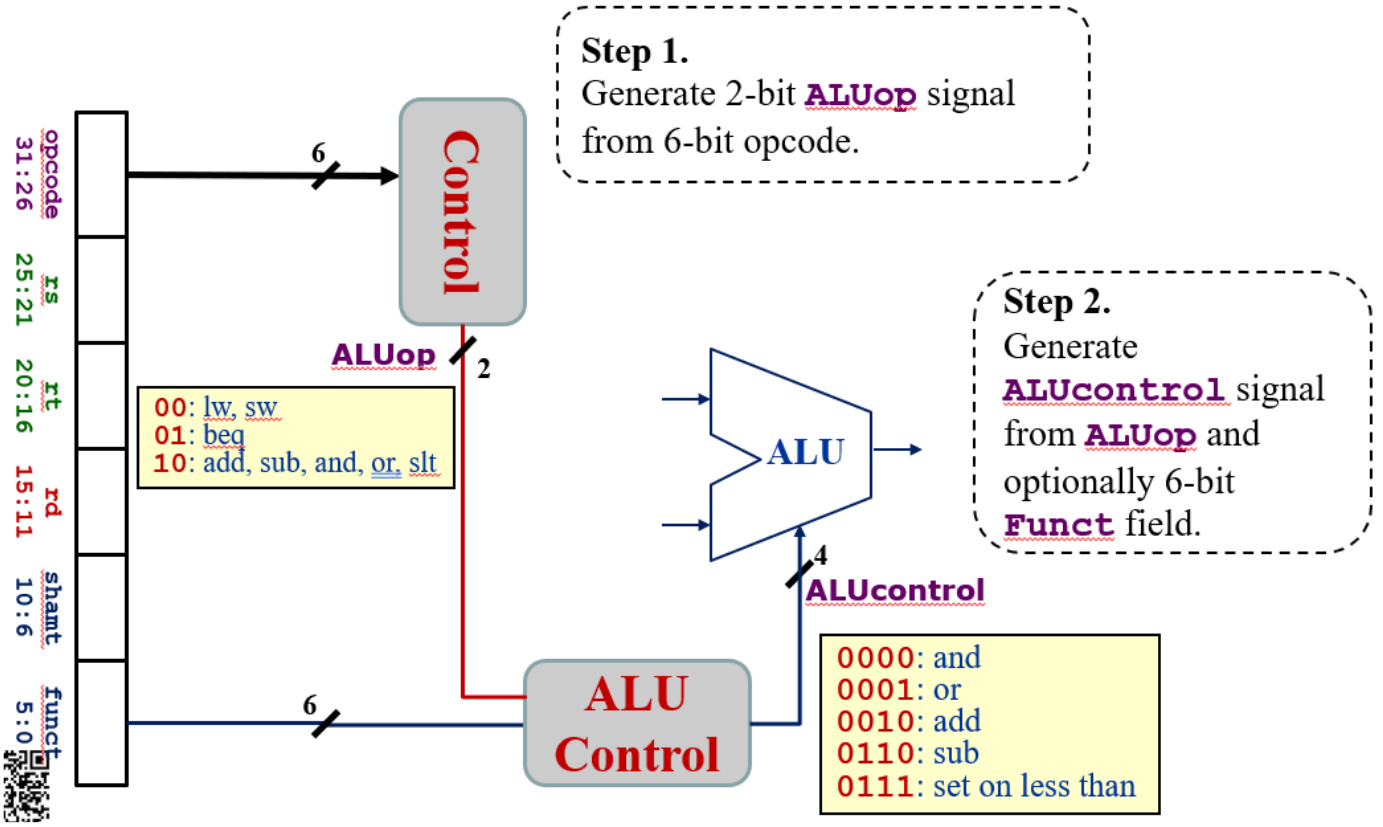
为什么需要 ALUop ：

引入 ALUop 作为一个中间步骤有几个好处：

1. **简化解码**：通过首先将 opcode 转换为一个更简单的 ALUop 信号，解码器可以在处理完整的 opcode 和 function code 之前，先进行一次“预解码”，这减少了同时需要考虑的变量数量。
2. **减少硬件复杂性**：直接解析整个 opcode 和 function code 可能需要很多逻辑门，而这种方法通过减少每个阶段需要的逻辑复杂性来减少所需的硬件。
3. **模块化设计**：这种分级方法允许设计者在不同的层次上考虑问题，可能使得测试和故障排除更加容易。

通过这种分步骤的方法，系统可以更有效地生成正确的 ALUcontrol 信号，即使在面对多种可能的操作和复杂的指令集时也是如此。这就是多级解码在实际应用中的一个例子。

8.5.4 Two-Level Implementation



8.5.5 Generating ALUcontrol Signal

Opcode	ALUop	Instruction Operation	Funct field	ALU action	ALU control
lw	00	load word	xxxxxx	add	0010
sw	00	store word	xxxxxx	add	0010
beq	01	branch equal	xxxxxx	subtract	0110
R-type	10	add	10 0000	add	0010
R-type	10	subtract	10 0010	subtract	0110
R-type	10	AND	10 0100	AND	0000
R-type	10	OR	10 0101	OR	0001
R-type	10	set on less than	10 1010	set on less than	0111

Instruction Type	ALUop
lw / sw	00
beq	01
R-type	10

ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR

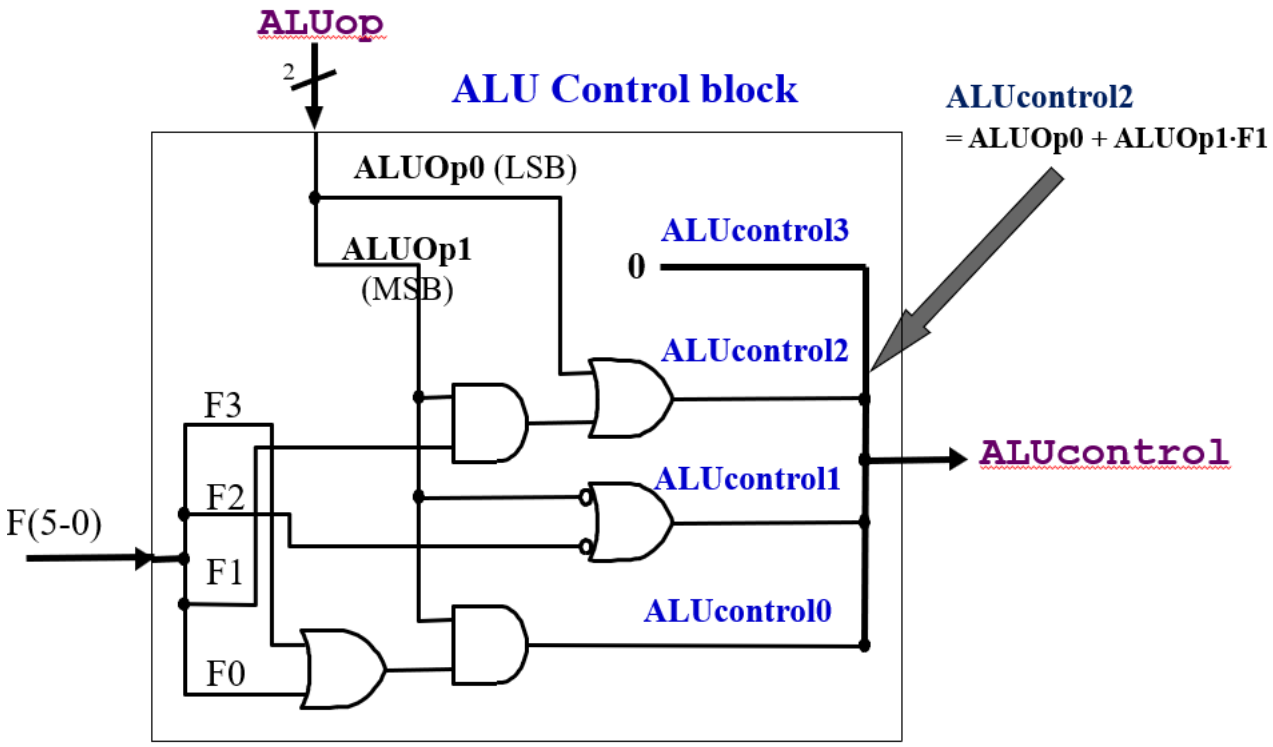
Generation of 2-bit ALUop signal will be discussed later

8.5.6 Design of ALU Control Unit

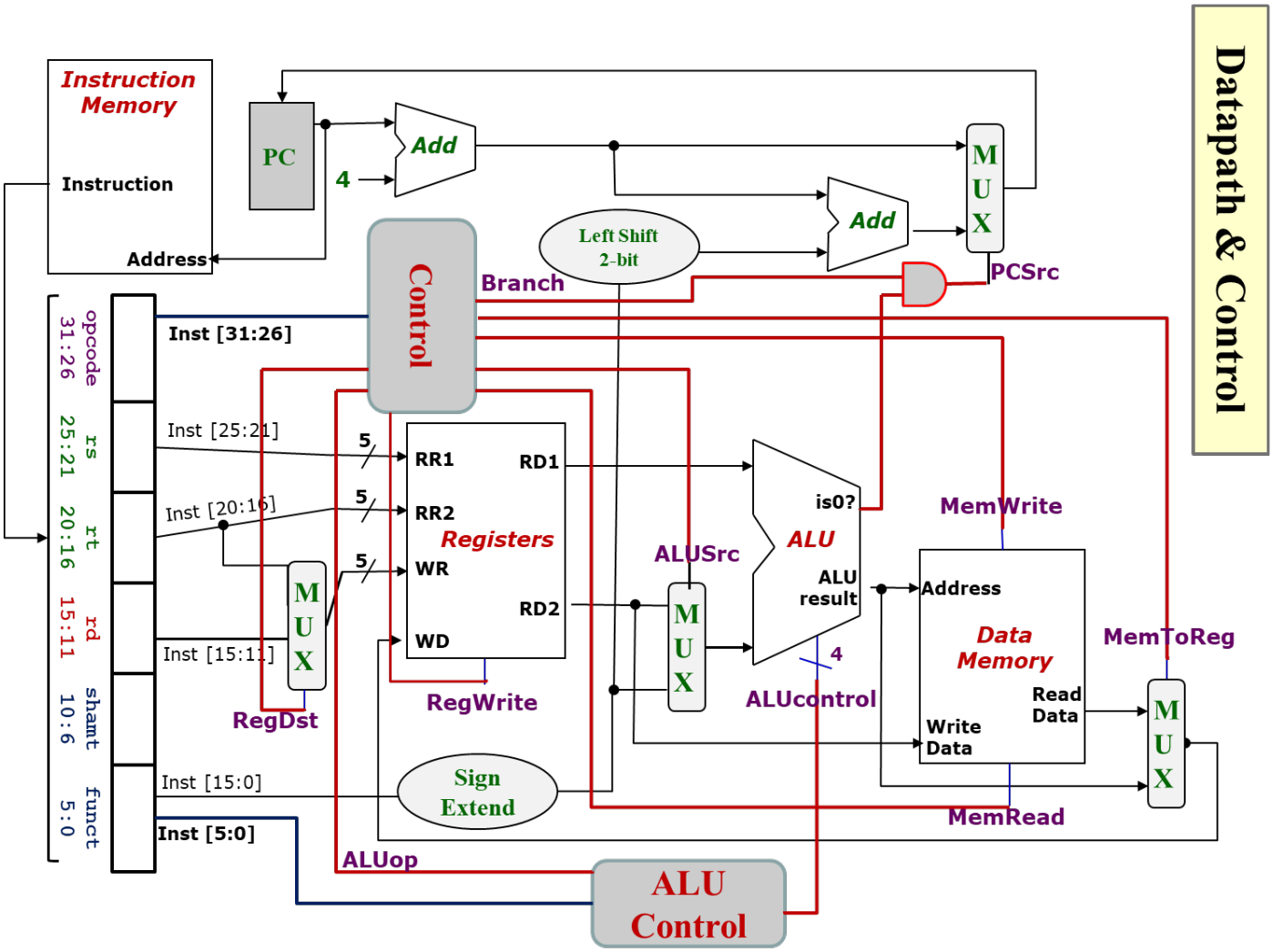
- Input: 6-bit `funct` field and 2-bit `ALUop`
- Output: 4-bit `ALUcontrol`

	ALUop		Funct Field (F[5:0] == Inst[5:0])						ALU control
	MSB	LSB	F5	F4	F3	F2	F1	F0	
lw	0	0	X	X	X	X	X	X	0 0 1 0
sw	0	0	X	X	X	X	X	X	0 0 1 0
beq	~~0~~ X	1	X	X	X	X	X	X	0 1 1 0
add	1	~~0~~ X	~~1~~ X	~~0~~ X	0	0	0	0	0 0 1 0
sub	1	~~0~~ X	~~1~~ X	~~0~~ X	0	0	1	0	0 1 1 0
and	1	~~0~~ X	~~1~~ X	~~0~~ X	0	1	0	0	0 0 0 0
or	1	~~0~~ X	~~1~~ X	~~0~~ X	0	1	0	1	0 0 0 1
slt	1	~~0~~ X	~~1~~ X	~~0~~ X	1	0	1	0	0 1 1 1

- Simple combinational logic



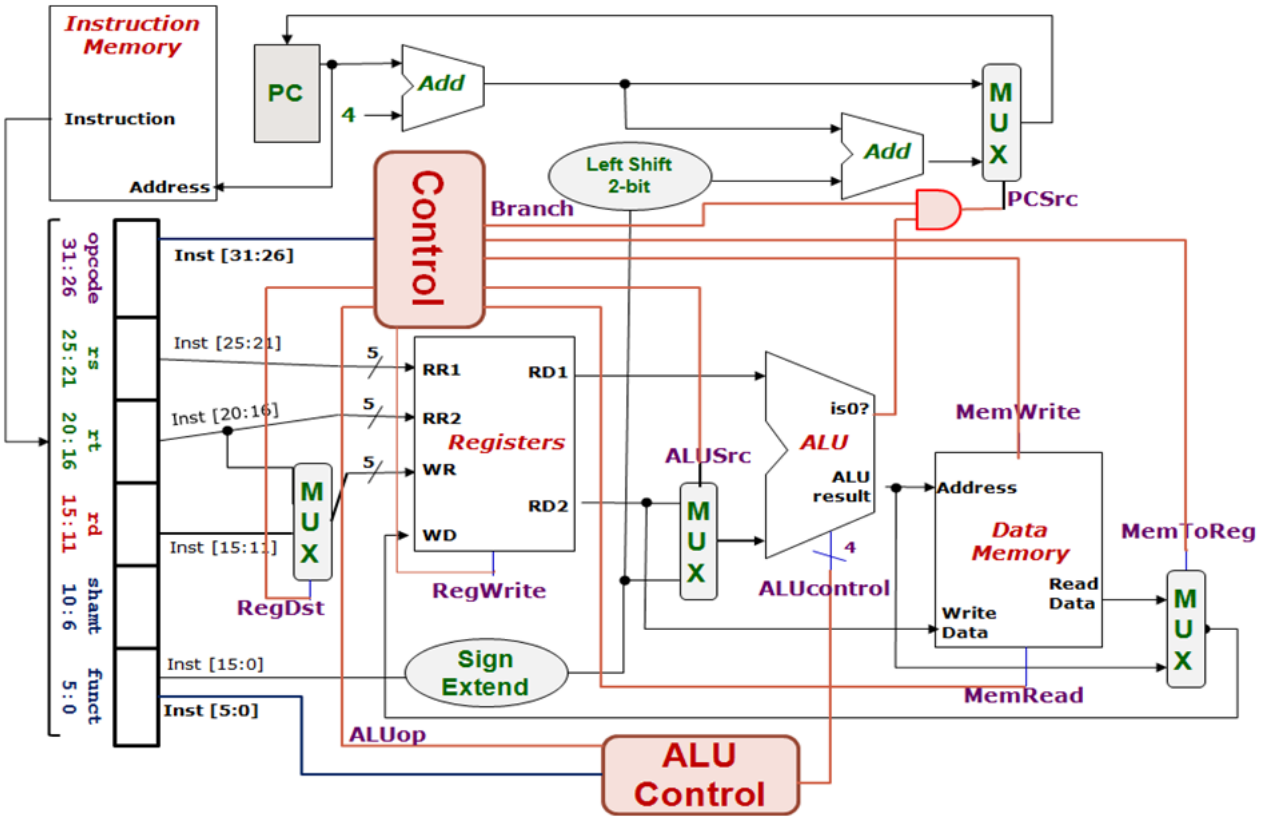
8.5.7 Summary



- Typical digital design steps:
  - Fill in truth table
    - Input: `opcode`
    - Output: Various control signals as discussed
  - Derive simplified expression for each signal

8.5.8 Control Design: Outputs

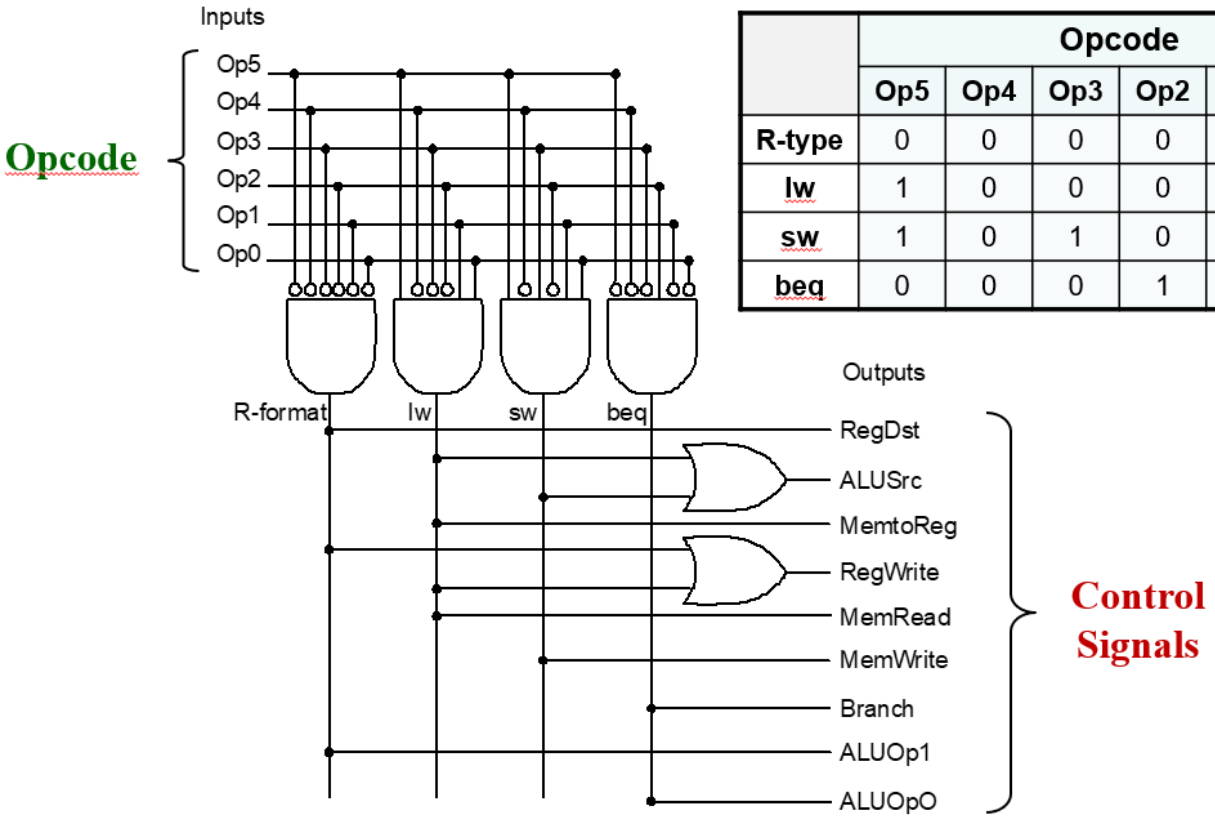
	RegDst	ALUSrc	MemTo Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop	
								op1	op0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1



8.5.9 Control Design: Inputs

	Opcode (Op[5:0] == Inst[31:26])						
	Op5	Op4	Op3	Op2	Op1	Op0	Value in Hexadecimal
R-type	0	0	0	0	0	0	0
lw	1	0	0	0	1	1	23
sw	1	0	1	0	1	1	2B
beq	0	0	0	1	0	0	4

8.5.10 Combinational Circuit Implementation



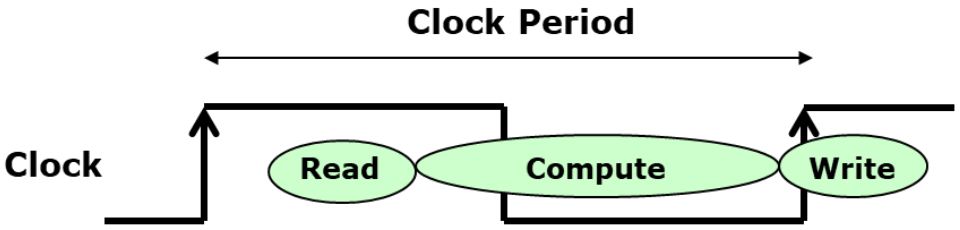
	Opcode					
	Op5	Op4	Op3	Op2	Op1	Op0
R-type	0	0	0	0	0	0
lw	1	0	0	0	1	1
sw	1	0	1	0	1	1
beq	0	0	0	1	0	0

8.6 Instruction Execution

Instruction Execution =

1. Read contents of one or more storage elements (register/memory)
2. Perform computation through some combinational logic
3. Write results to one or more storage elements (register/memory)

All these performed within a clock period



Don’t want to read a storage element when it is being written.

8.6.1 Single Cycle Implementation: Shortcoming

Calculate cycle time assuming negligible delays: memory (2ns), ALU/adders (2ns), register file access (1ns)

Instruction	Inst Mem	Reg read	ALU	Data Mem	Reg write	Total
ALU	2	1	2		1	6
lw	2	1	2	2	1	8
sw	2	1	2	2		7
beq	2	1	2			5

All instructions take as much time as the slowest one (i.e. load)

→ Long cycle time for each instruction

这段内容在讨论单周期处理器实现的一个重要限制。在单周期（Single Cycle）处理器架构中，处理器在一个时钟周期内完成整个指令。时钟周期的长度由执行所有指令所需的最长路径决定。这里提供了一个简化的例子来说明这个概念。

首先，表格列出了不同类型指令在各个阶段所需的时间。这些阶段包括指令内存访问（从内存中获取指令）、寄存器文件读取（从寄存器读取数据）、ALU操作（执行算术或逻辑操作）、数据内存访问（对内存进行加载或存储操作）和寄存器写入（将结果写回寄存器）。每个阶段的延迟被假设为一个确定的值，例如内存访问为2纳秒，ALU和加法器操作为2纳秒，寄存器文件访问为1纳秒。

现在，考虑到不同指令的需求，表中展示了它们各自所需的总时间。例如，“lw”（加载字）指令需要8纳秒，因为它涉及所有的步骤：指令内存、寄存器读取、ALU操作、数据内存和寄存器写入。

然而，单周期处理器的一个关键缺点是所有的指令必须在一个单一的、固定长度的时钟周期内完成。这个周期的长度由最慢的指令（在这个例子中是“lw”指令，需要8纳秒）决定。即使其他指令（如“beq”或ALU操作）可以更快地完成，时钟周期也不能更短，因为它必须足够长以容纳最慢的指令。结果是，所有的指令都会受到最慢指令的“拖累”，导致整体性能的下降。

总结一下，单周期实现的主要缺点是：

1. 时钟周期时间由最慢的指令决定，导致效率低下。
2. 更快的指令不得不等待，不能立即释放系统资源，从而减少了处理器的吞吐量。
3. 不能充分利用可能的并行性，因为下一指令直到当前指令完成之后才开始，即使它所需的资源已经可用。

这些限制促使了其他类型的CPU设计，例如多周期和流水线架构，这些架构可以更有效地处理指令集中的时间差异。

8.6.2 Solution #1: Multicycle Implementation

Break up the instructions into execution steps:

1. Instruction fetch
2. Instruction decode and register read
3. ALU operation
4. Memory read/write
5. Register write

Each execution step takes one clock cycle

- Cycle time is much shorter, i.e., clock frequency is much higher

Instructions take variable number of clock cycles to complete execution

8.6.3 Pipelining

Break up the instructions into execution steps one per clock cycle

Allow different instructions to be in different execution steps simultaneously

> “Pipelining”是计算机架构中提高处理器性能的关键技术之一。它不是通过“切片”来实现的，而是通过将指令处理过程分解为几个连续的步骤或阶段，每个步骤在各自的硬件中执行。这些步骤或阶段是按照顺序排列的，每个阶段完成一个特定的部分操作。通过这种方式，处理器可以在不同阶段同时处理多条指令。
>
> **实现流水线的步骤通常包括：**
>
> 1. **指令取回（Instruction Fetch，IF）** - 处理器从内存中读取下一条要执行的指令。
> 2. **指令译码（Instruction Decode，ID）** - 解码器将二进制指令解码为处理器可以理解的指令，并确定需要使用的数据。
> 3. **执行（Execute，EX）** - ALU（算术逻辑单元）执行所需的计算，比如加法、减法、乘法等。
> 4. **内存访问（Memory Access，MEM）** - 如果指令需要，处理器会在这一步读取或写入数据到内存。
> 5. **写回（Write-back，WB）** - 处理器将执行结果写回到寄存器。
>
> 在流水线处理中，上述每个步骤都在各自的时钟周期内发生，并且它们是重叠的。例如，在一个给定的时钟周期内，一条指令可能处于“执行”阶段，而另一条指令可能处于“指令取回”阶段。这就允许在每个时钟周期内启动一条新的指令，大大提高了处理器的吞吐量和效率。
>
> 这种方法的效率很大程度上依赖于指令和流水线阶段的划分能否使得每个阶段都尽可能短且均衡，从而避免某个阶段过长而造成的瓶颈。

9 - Pipelining

9.1 Introduction

Pipelining doesn't help latency of single task:

- It helps the throughput of the entire wordload

Multiple tasks operating simultaneously using different resources

Possible delays:

- Pipeline rate limited by slowest pipeline stage
- Stall of dependencies

9.2 MIPS Pipeline Stages

Five execution stages:

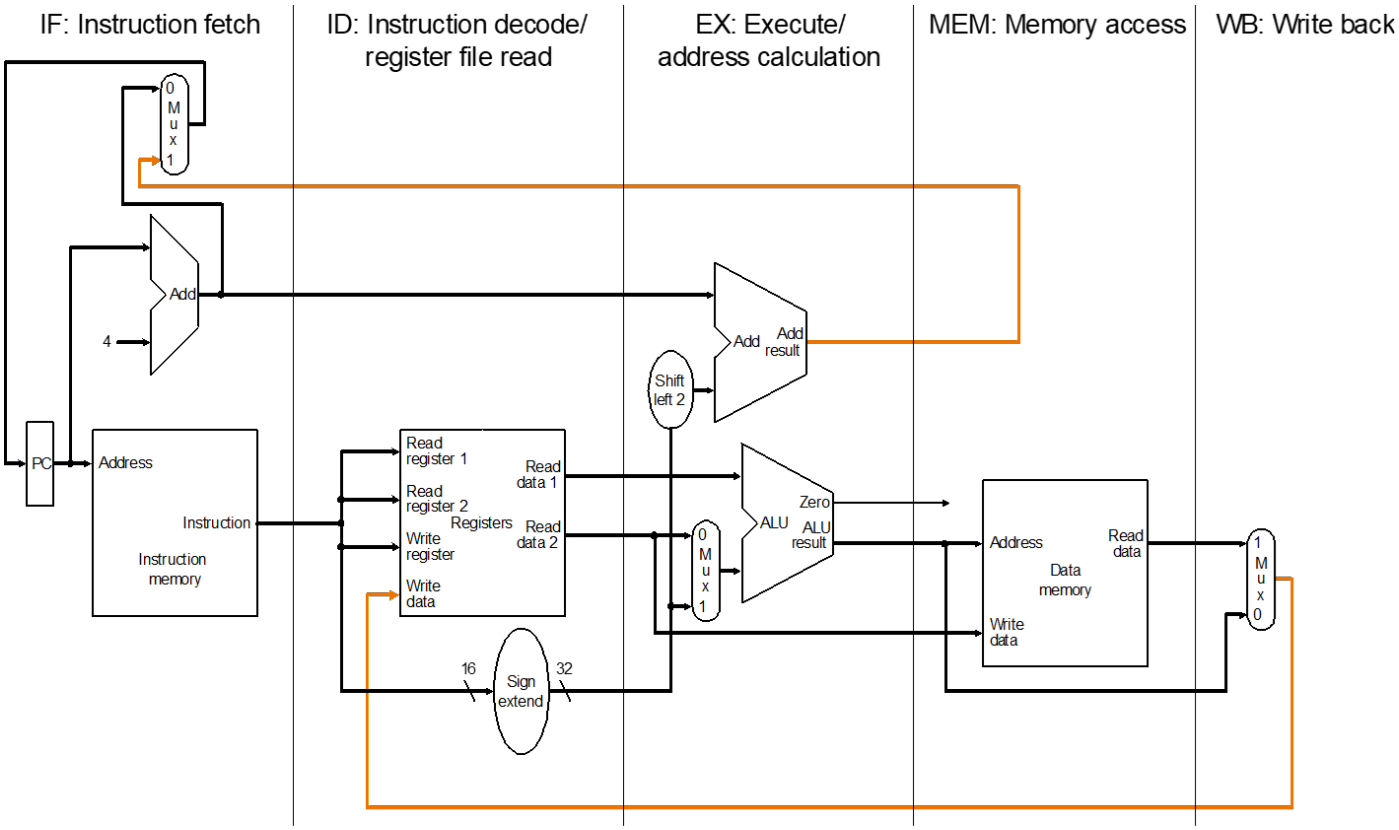
1. `IF` : Instruction Fetch
2. `ID` : Instruction Decode and Register Read
3. `EX` : Execute an operation or calculate an address
4. `MEM` : Access an operand in data memory
5. `WB` : Write Back the result into a register

Idea:

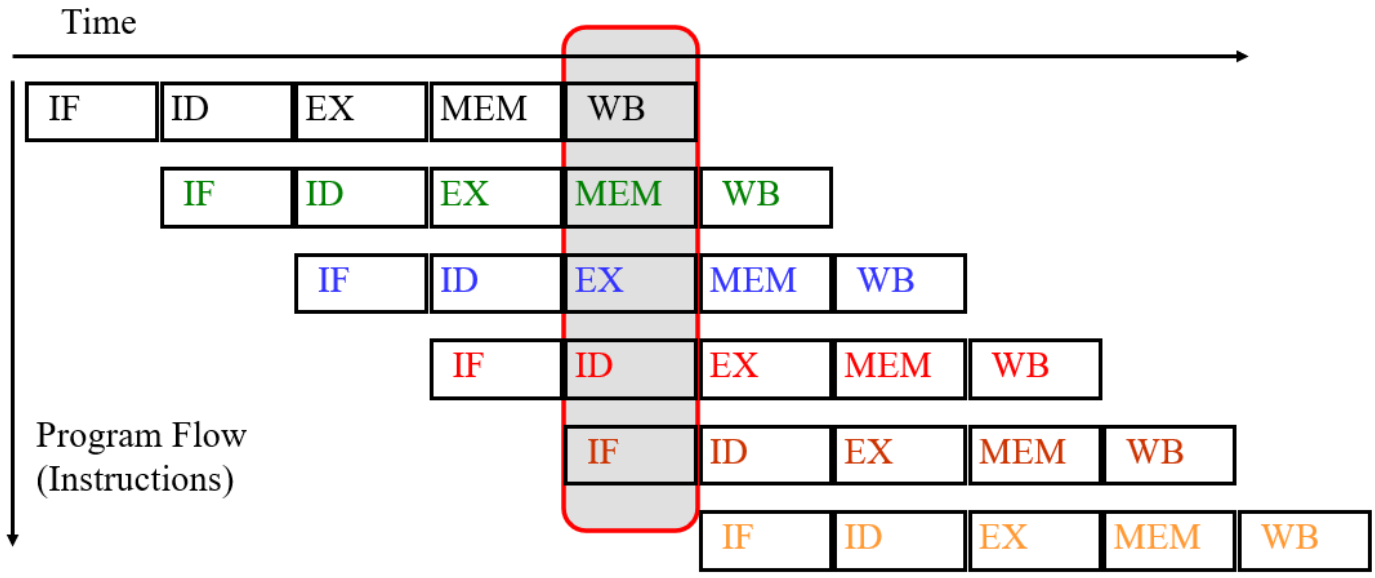
- Each execution stage takes 1 clock cycle
- General flow of data is from one stage to the next

Exceptions:

- Update of PC and write back of register file



9.2.1 Pipelined Execution: Illustration



9.3 Pipeline Datapath

Single-cycle implementation:

- Upload all state elements (PC, register file, data memory) at the end of a clock cycle

Pipelined implementation:

- One cycle per pipeline stage
- Data required for each stage needs to be stored separately

单周期实现：

在单周期处理器中，每个指令从开始到结束都在一个时钟周期内完成。因此，所有的状态元素（如程序计数器（PC）、寄存器文件、数据内存等）都是在时钟周期的末尾更新的。这意味着，在下一个周期开始之前，CPU执行完整条指令的所有步骤。这种方法简单、清晰，但速度受限于最慢的指令，因为所有指令都必须在同一个周期长度内完成。

流水线实现：

与单周期不同，流水线处理器将指令执行划分为几个阶段，每个阶段在一个时钟周期内完成一部分任务。这样做的目的是让不同指令的不同部分能够并行执行，从而在给定时间内完成更多的指令。

现在，关于为什么每个阶段需要的数据需要分别存储，这里有几点关键原因：

1. **防止数据冲突和冒险**：在流水线中，多条指令会重叠执行。一条指令的某个阶段可能需要使用前一条指令的结果。如果所有数据都存储在同一位置，一条指令的输出可能会覆盖另一条指令的关键数据，导致错误。通过在每个阶段结束时存储数据，我们可以保证每条指令都能访问其所需的正确数据，而不会被其他同时执行的指令干扰。
2. **阶段间同步**：由于每个阶段都在自己的时钟周期内独立操作，所以必须有一种机制确保数据在正确的时间传输到下一个阶段。这意味着每个阶段结束时，其输出数据必须被存储在一个地方，以便下一个阶段在下一个时钟周期开始时可以使用。这通常是通过在各个阶段之间使用寄存器（被称为流水线寄存器）来实现的。
3. **增强处理能力**：将每个阶段需要的数据分开存储，意味着当一个阶段正在处理一条指令时，其他阶段可以同时读取和处理来自/要传递到其他指令的数据。这消除了处理过程中的闲置时间，允许处理器更快地执行指令序列。

简而言之，流水线实现通过在每个阶段的结尾单独存储状态和数据，使得多条指令可以同时且高效地在不同阶段执行，增加了整个处理器的吞吐量和效率。而这种分阶段存储的需求来源于并行指令执行过程中对数据完整性和正确同步的基本需求

Data used by subsequent instructions:

- Store in programmer-visible state elements: `PC`, register file and memory

Data used by same instruction in later pipeline stages:

- Additional registers in datapath called pipeline registers
- `IF/ID`: register between `IF` and `ID`
- `ID/EX`: register between `ID` and `EX`
- `EX/MEM`: register between `EX` and `MEM`
- `MEM/WB`: register between `MEM` and `WB`

在流水线处理器架构中，如何管理和存储在各个阶段中产生并且在后续阶段中需要使用的数据。为了保持处理器的高效运行，防止数据冲突和数据冒险，流水线架构使用了特殊的寄存器，称为流水线寄存器。下面详细解释这段话的内容。

程序可见状态元素：

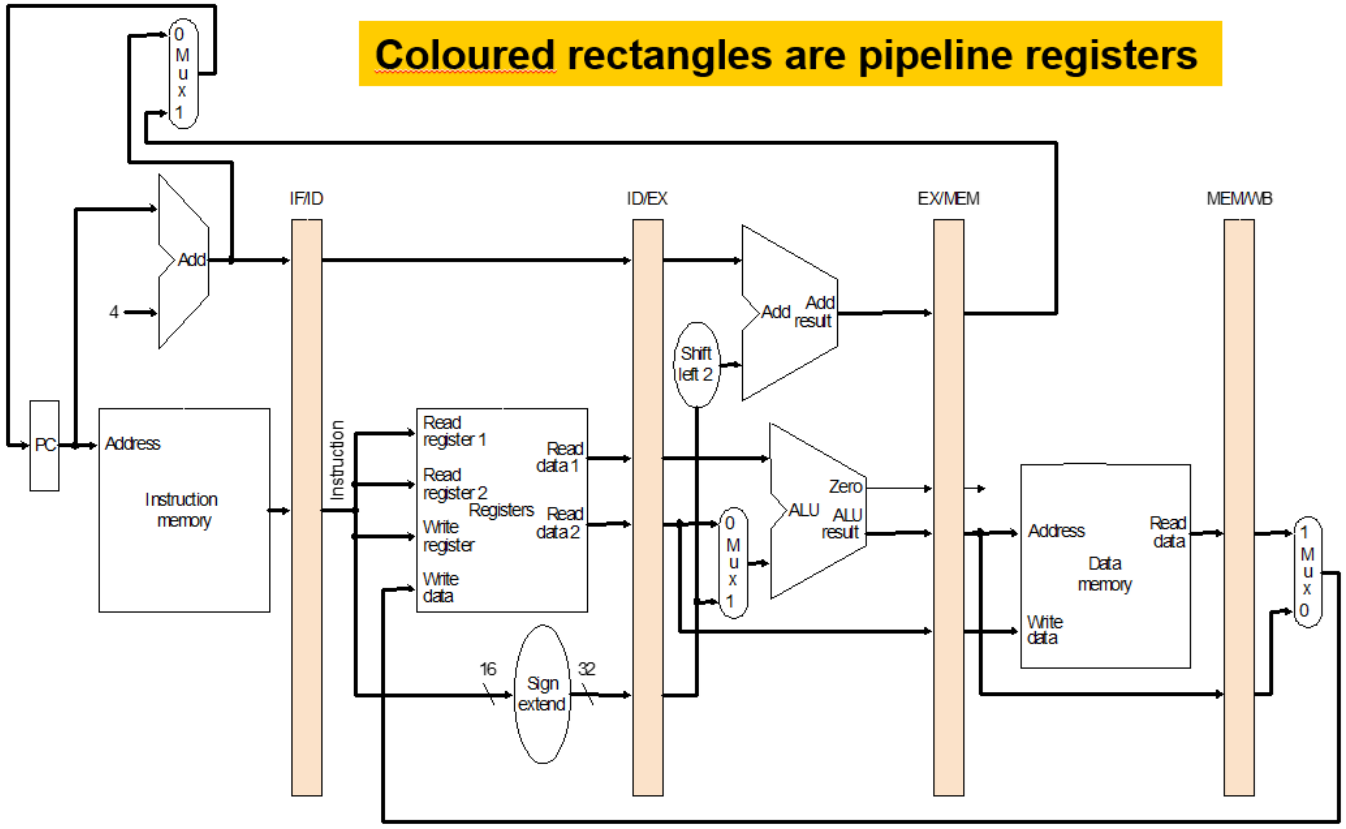
- **程序计数器（PC）**：它存储的是下一条要执行的指令的地址。
- **寄存器文件**：这是一组寄存器，用于存储在指令执行过程中需要的数据。
- **内存**：这是一个更大的数据存储区域，用于存储指令以及指令操作的数据。

这些元素对程序员是可见的，因为他们在编写程序时可以直接或间接地操作这些元素。

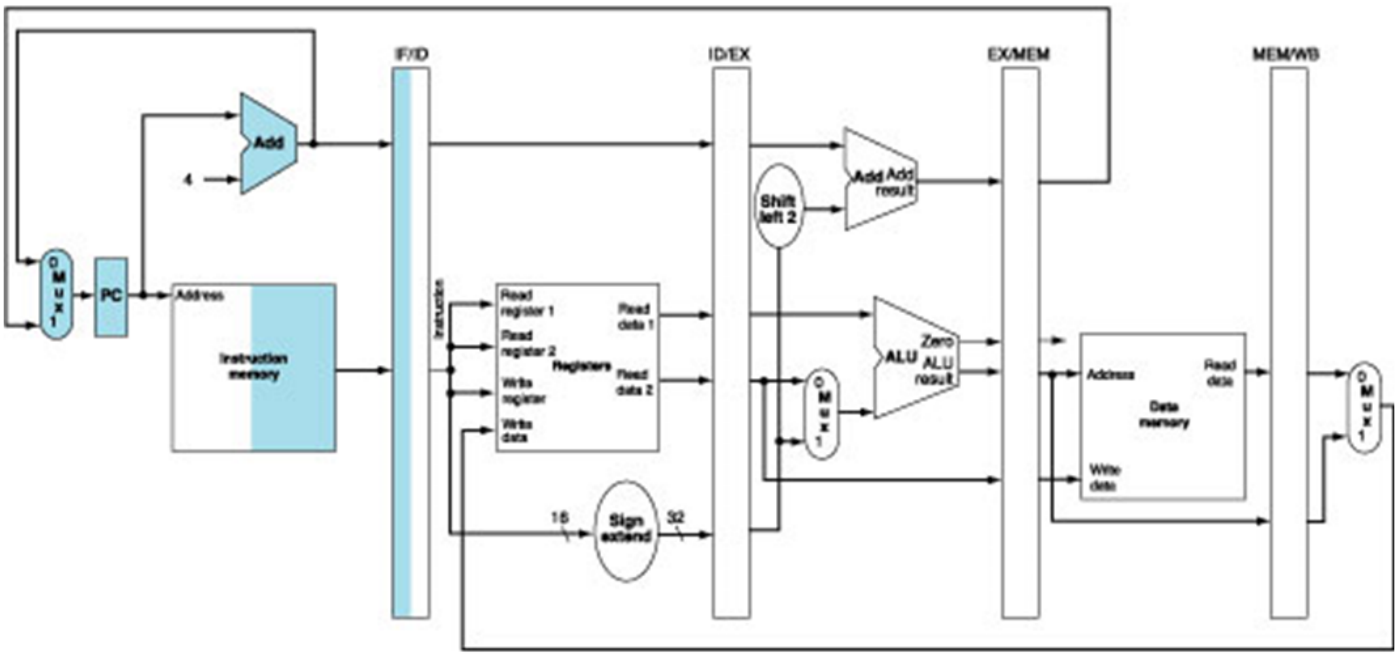
流水线寄存器：

流水线寄存器是在流水线的各个阶段之间放置的寄存器。它们在每个时钟周期结束时捕获并存储当前阶段的输出，这些输出数据将在下一个时钟周期中的下一个阶段被使用。这样做是为了保证即使在下一个时钟周期中当前阶段有新的数据产生，也不会影响下一阶段需要的数据。

- **IF/ID**：这个寄存器存储的是从“指令取指”阶段（IF）到“指令译码”阶段（ID）的数据。这包括被取出的指令以及程序计数器（PC）的值。
- **ID/EX**：这个寄存器在“指令译码”阶段（ID）和“执行”阶段（EX）之间。它存储的数据通常包括译码后的指令信息，操作数，以及要执行的下一操作的必要信息。
- **EX/MEM**：这个寄存器连接“执行”阶段（EX）和“访问内存”阶段（MEM）。它会存储ALU的操作结果，以及对内存的任何访问指令（如果有的话）。
- **MEM/WB**：这个寄存器位于“访问内存”阶段（MEM）和“写回”阶段（WB）之间。它会存储从内存读取的数据（如果指令涉及内存读取的话）和/或ALU的结果，这些数据需要写回到寄存器文件中。



9.3.1 IF Stage

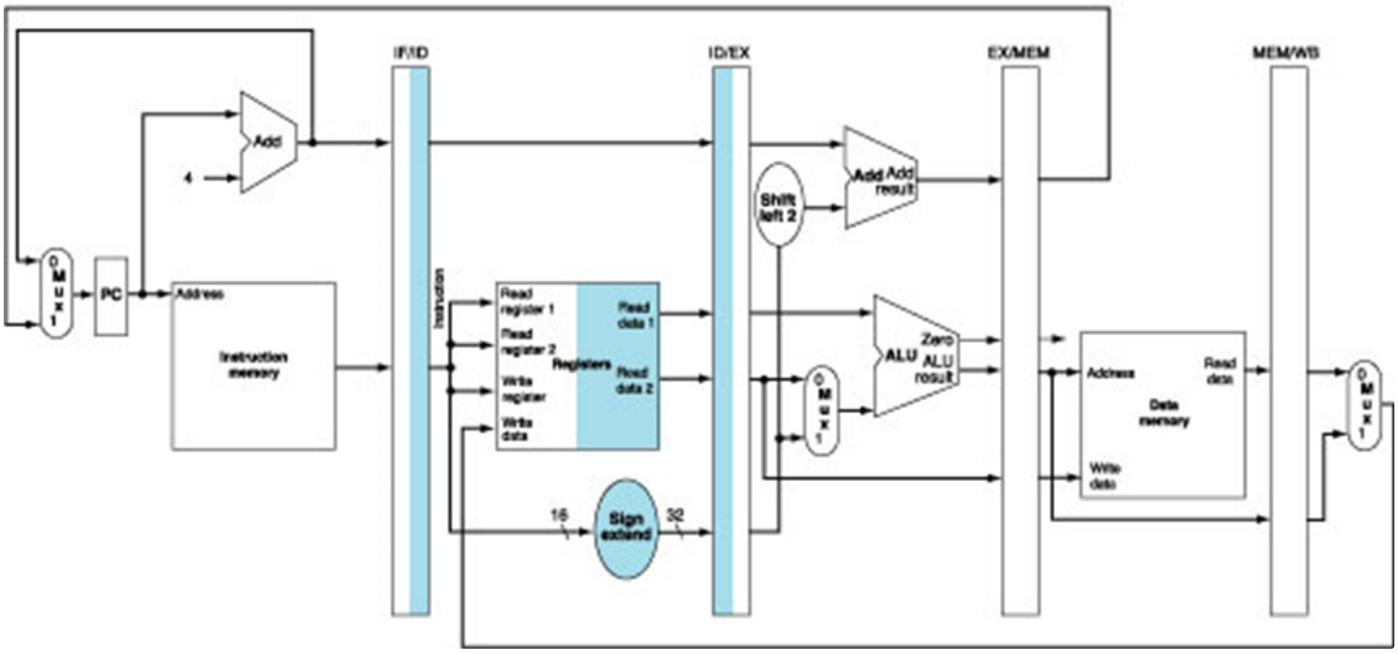


At the end of a cycle, `IF/ID` receives (stores):

- Instruction read from InstructionMemory[PC]
- PC+4

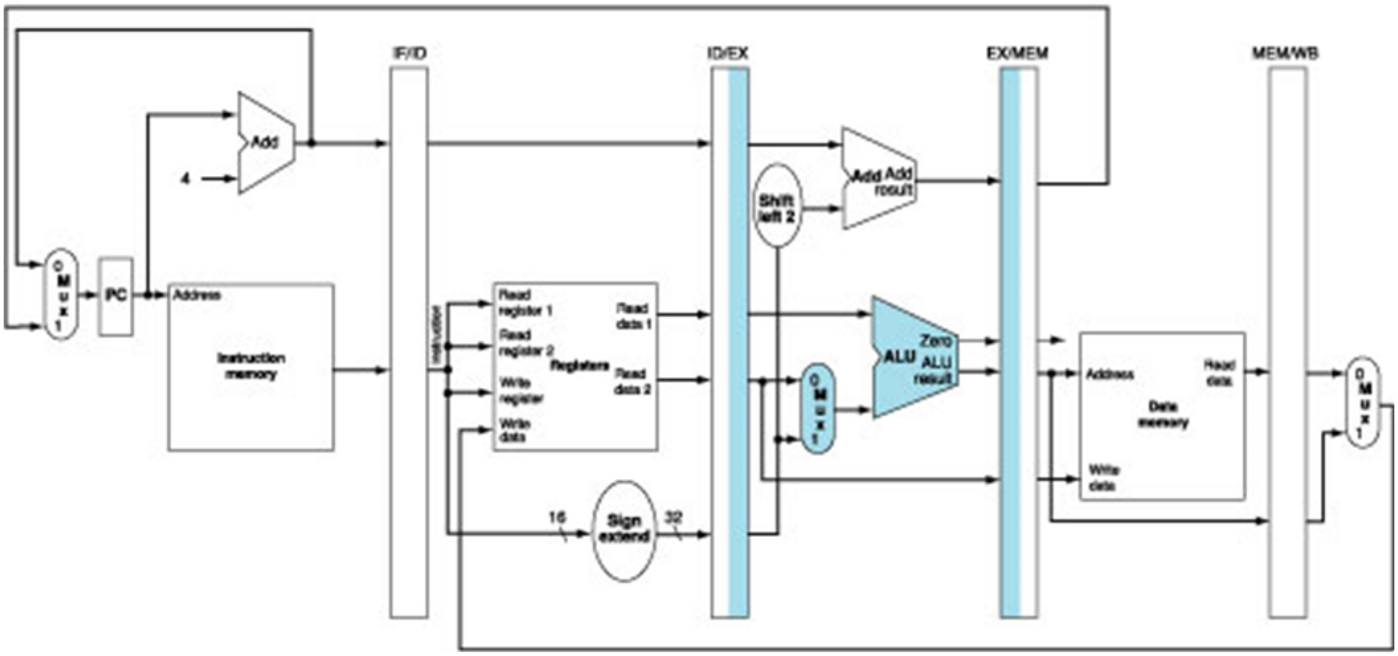
PC+4 also connected to one of the MUX's inputs

9.3.2 `ID` Stage



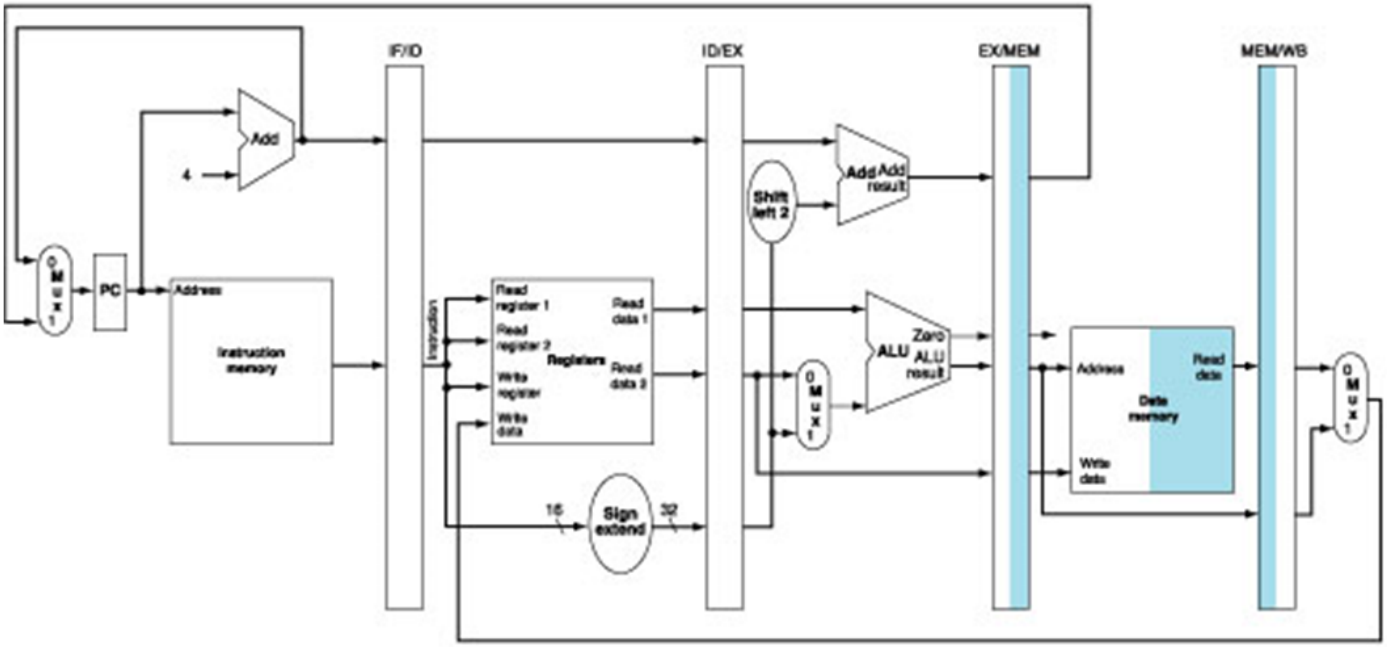
At the beginning of a cycle `IF/ID` register supplies:	At the end of a cycle `ID/EX` receives:
❖ Register numbers for reading two registers ❖ 16-bit offset to be sign-extended to 32-bit	❖ Data values read from register file ❖ 32-bit immediate value ❖ **`PC + 4`**

9.3.3 `EX` Stage



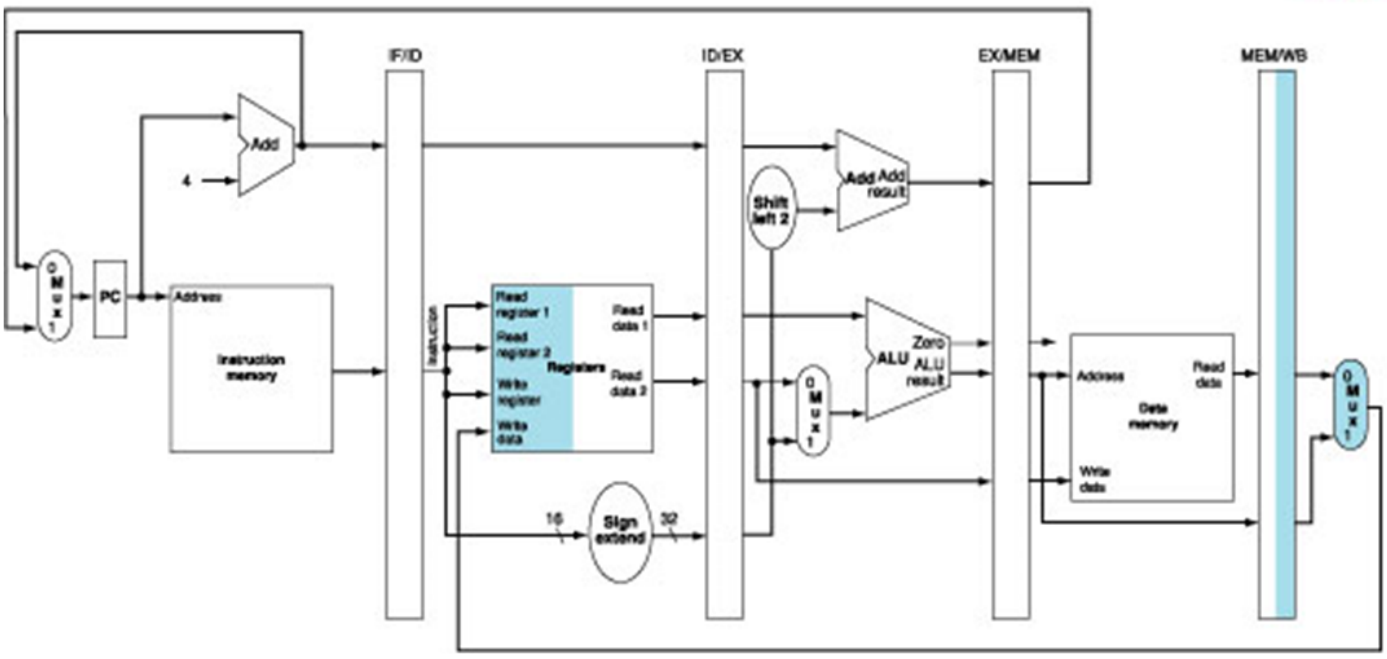
At the beginning of a cycle `ID/EX` register supplies:	At the end of a cycle `EX/MEM` receives:
❖ Data values read from register file ❖ 32-bit immediate value ❖ **`PC + 4`**	❖ (PC + 4) + (Immediate x 4) ❖ ALU result ❖ **`isZero?`** signal ❖ Data Read 2 from register file

9.3.4 `MEM` Stage



At the beginning of a cycle **EX/MEM** register supplies:	At the end of a cycle **MEM/WB** receives:
❖ (PC + 4) + (Immediate x 4) ❖ ALU result ❖ **`isZero?`** signal ❖ Data Read 2 from register file	❖ ALU result ❖ Memory read data

9.3.5 WB Stage



At the beginning of a cycle MEM/WB register supplies:	At the end of a cycle
❖ ALU result ❖ Memory read data	❖ Result is written back to register file (if applicable) ❖ There is a bug here…….

1. **取指阶段（IF - Instruction Fetch）：**
   - **提供的数据：** 这个阶段不接收来自上一个流水线寄存器的数据，而是从程序计数器（PC）获取当前指令的地址。
   - **接收的数据（进入 IF/ID 寄存器）：** 该阶段从内存中获取指令，并将指令本身以及下一条指令的地址（通常是当前 PC + 4）提供给 IF/ID 寄存器。
2. **指令译码/寄存器读取阶段（ID - Instruction Decode）：**
   - **提供的数据（来自 IF/ID 寄存器）：** 该阶段接收上一阶段取得的指令和下一条指令的地址。
   - **接收的数据（进入 ID/EX 寄存器）：** 该阶段解码指令，读取必要的寄存器，并将以下信息传递给下一阶段：解码的操作码和操作数、从寄存器文件中读取的数据、即将执行的指令的控制信号（例如，ALU应执行的操作，是否需要访问内存等）。
3. **执行/地址计算阶段（EX - Execution）：**
   - **提供的数据（来自 ID/EX 寄存器）：** 此阶段接收解码的指令数据、操作数、以及控制信号。
   - **接收的数据（进入 EX/MEM 寄存器）：** ALU在这里执行计算或地址计算。该阶段将计算结果、任何要写入的值（对于存储指令）、计算出的内存地址（如果适用）以及控制信号传递给下一阶段。
4. **内存访问阶段（MEM - Memory Access）：**
   - **提供的数据（来自 EX/MEM 寄存器）：** 这个阶段接收来自ALU的计算结果，内存地址，要写入的值，以及控制信号。
   - **接收的数据（进入 MEM/WB 寄存器）：** 根据指令的需要，可能会从内存读取数据或向内存写入数据。它将读取的数据（如果有）、之前ALU的计算结果、以及控制信号传递到下一阶段。
5. **写回阶段（WB - Write-Back）：**
   - **提供的数据（来自 MEM/WB 寄存器）：** 此阶段接收可能从内存中读取的数据、ALU的计算结果，以及控制信号，确定是否需要将结果写回寄存器文件。
   - **接收的数据：** 在此阶段，数据被写回到寄存器文件中，完成指令的执行。由于这是流水线的最后一个阶段，因此不需要再将数据传递给另一个流水线寄存器。

9.3.6 Corrected Datapath

Observe the “Write register” number

- Supplied by the `IF/ID` pipeline register
- It is NOT the correct write register for the instruction now in `WB` stage

Solution:

- Pass “Write register” number from `ID/EX` through `EX/MEM` to `MEM/WB` pipeline register for use in `WB` stage

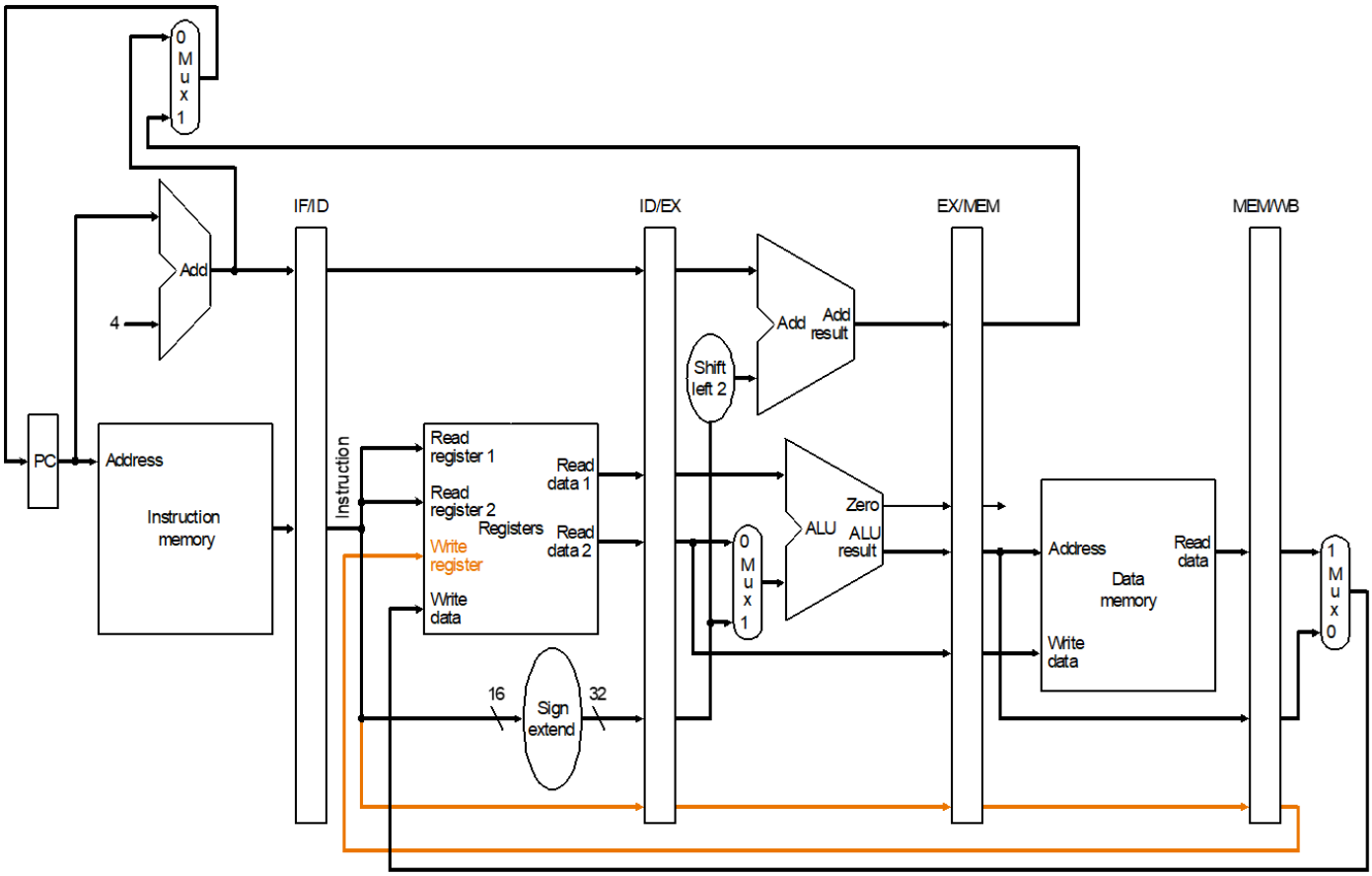
- i.e. let the “Write register” number follows the instruction through the pipeline until it is needed in `WB` stage

在流水线的设计中，每条指令通过各个阶段时，必须保持其相关数据的可用性，直到这些数据真正需要为止。观察到的问题是，“写寄存器”号（即将要写入的目标寄存器的地址）在流水线的某个点上不可用或不正确。在这种情况下，指令在“写回”（WB）阶段，需要知道数据应写回哪个寄存器，但是由于在流水线的早期阶段（IF/ID阶段）提供的“写寄存器”号，并没有随着指令一起传递，所以到达WB阶段时，它不是正确的寄存器号。

解决方案：

1. **传递“写寄存器”号码：** 解决方法是，从“指令译码”（ID）阶段开始，让“写寄存器”号随着指令一起通过流水线，经过“执行”（EX）和“内存访问”（MEM）阶段，最终到达“写回”（WB）阶段。这意味着在ID阶段确定的“写寄存器”号需要被存储并传递到流水线的后续阶段。
2. **通过流水线寄存器：** 为了实现这一点，系统引入了流水线寄存器，这些寄存器位于各个阶段之间。特别地，从ID/EX到EX/MEM，再到MEM/WB的流水线寄存器将携带这个“写寄存器”号。这保证了当指令到达WB阶段时，系统知道应该将数据写回哪个寄存器。

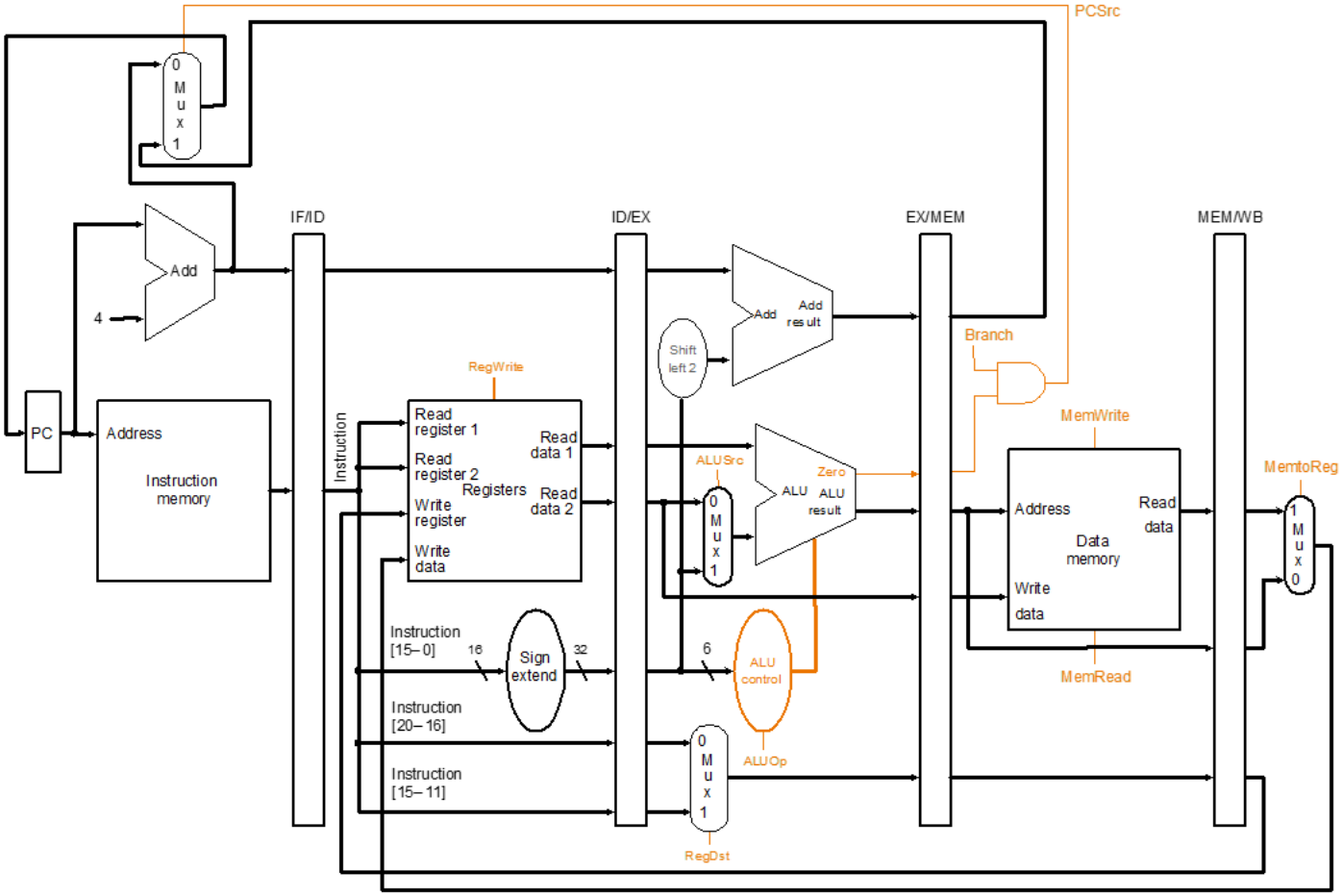
总的来说，这种更正确保了数据的准确性，并使流水线在处理每条指令时能够维持正确的状态。这是流水线设计中处理各种依赖关系的通用方法之一，确保数据和控制信息能够在需要时可用，从而避免错误和性能下降。



9.4 Pipeline Control

Main Idea

- Same control signals as single-cycle datapath
- Difference: Each control signal belongs to a particular pipeline stage



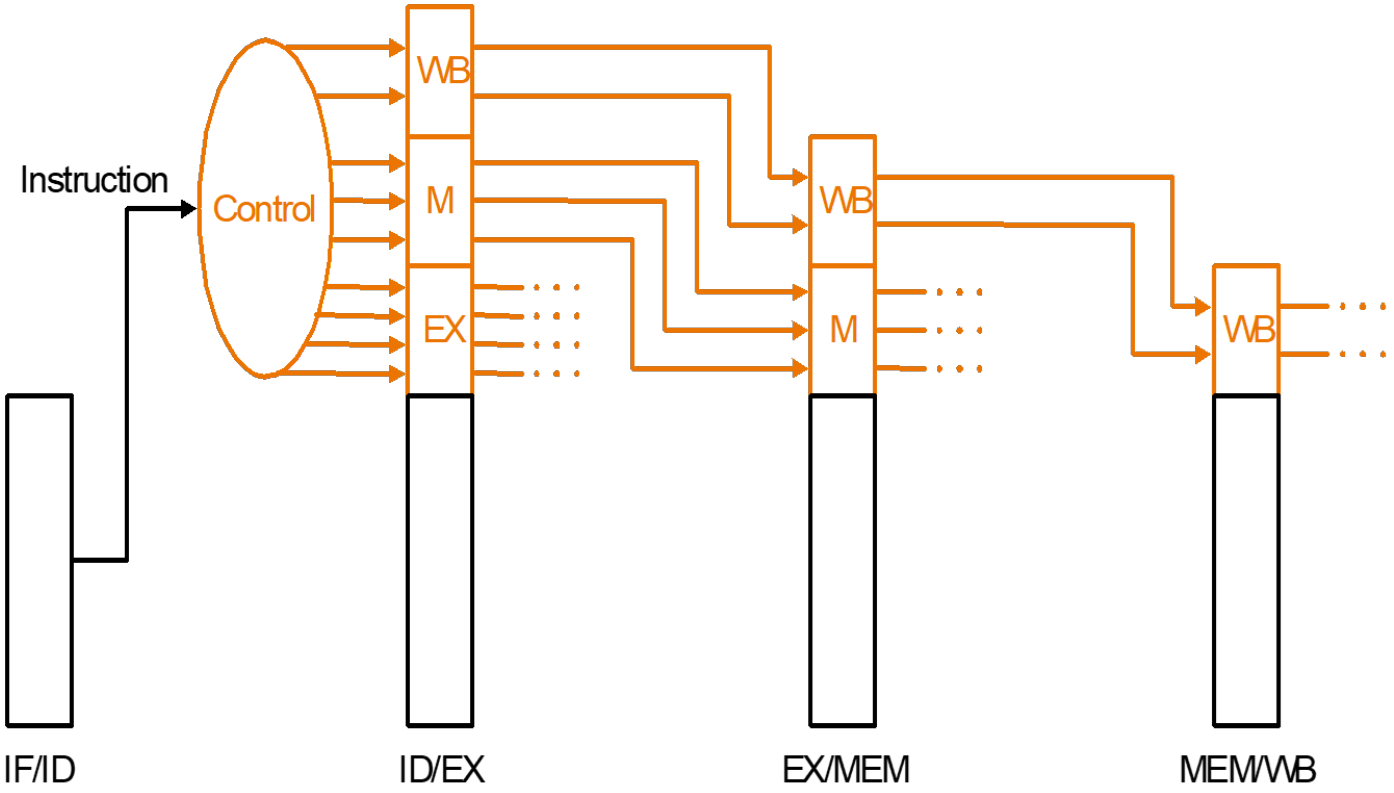
9.4.1 Grouping

Group control signals according to pipeline stage

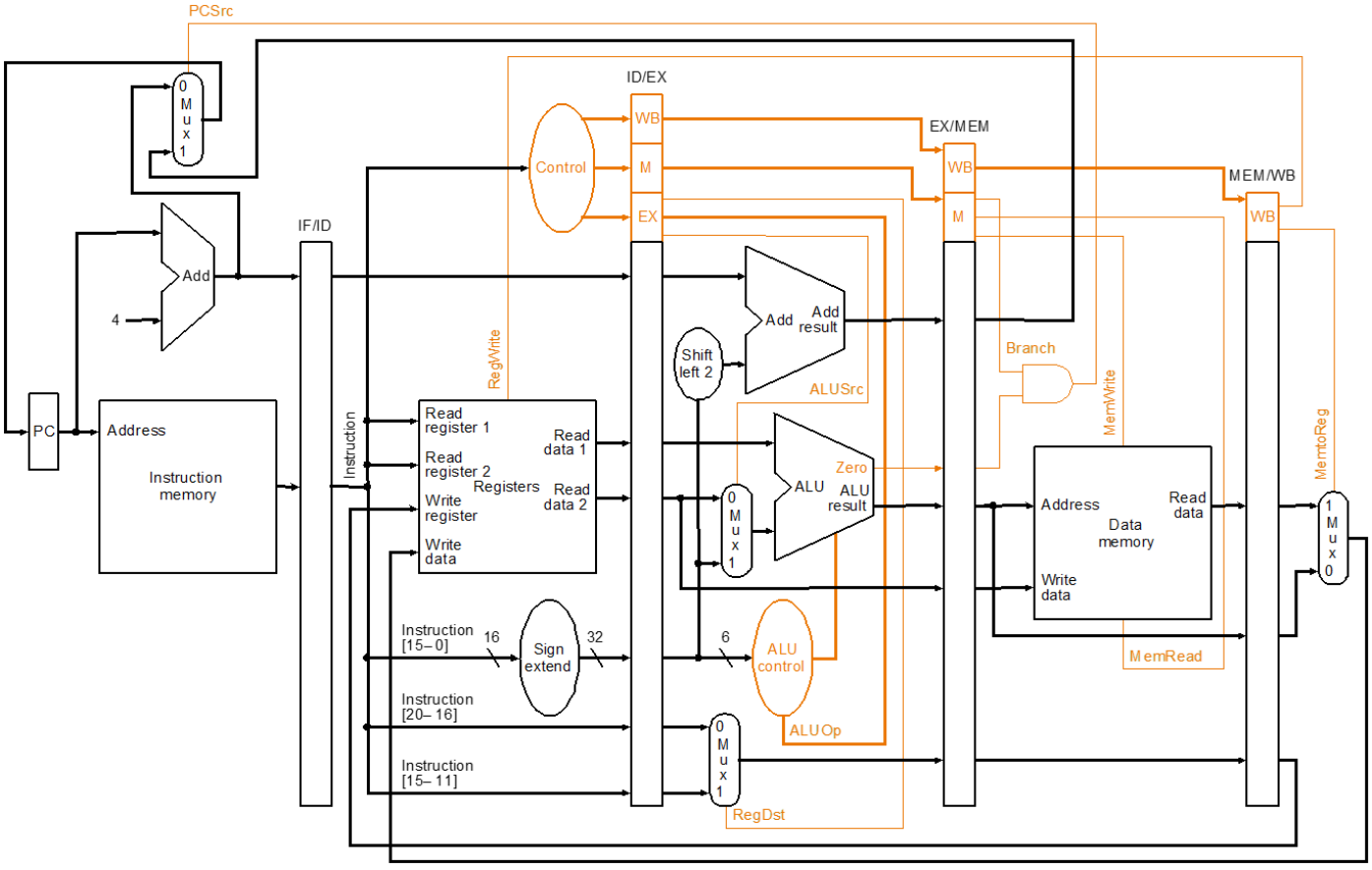
	RegDst	ALUSrc	MemTo Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop	
								op1	op0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

	EX Stage				MEM Stage			WB Stage	
	RegDst	ALUSrc	ALUop		Mem Read	Mem Write	Branch	MemTo Reg	Reg Write
			op1	op0					
R-type	1	0	1	0	0	0	0	0	1
lw	0	1	0	0	1	0	0	1	1
sw	X	1	0	0	0	1	0	X	0
beq	X	0	0	1	0	0	1	X	0

9.4.2 Implementation

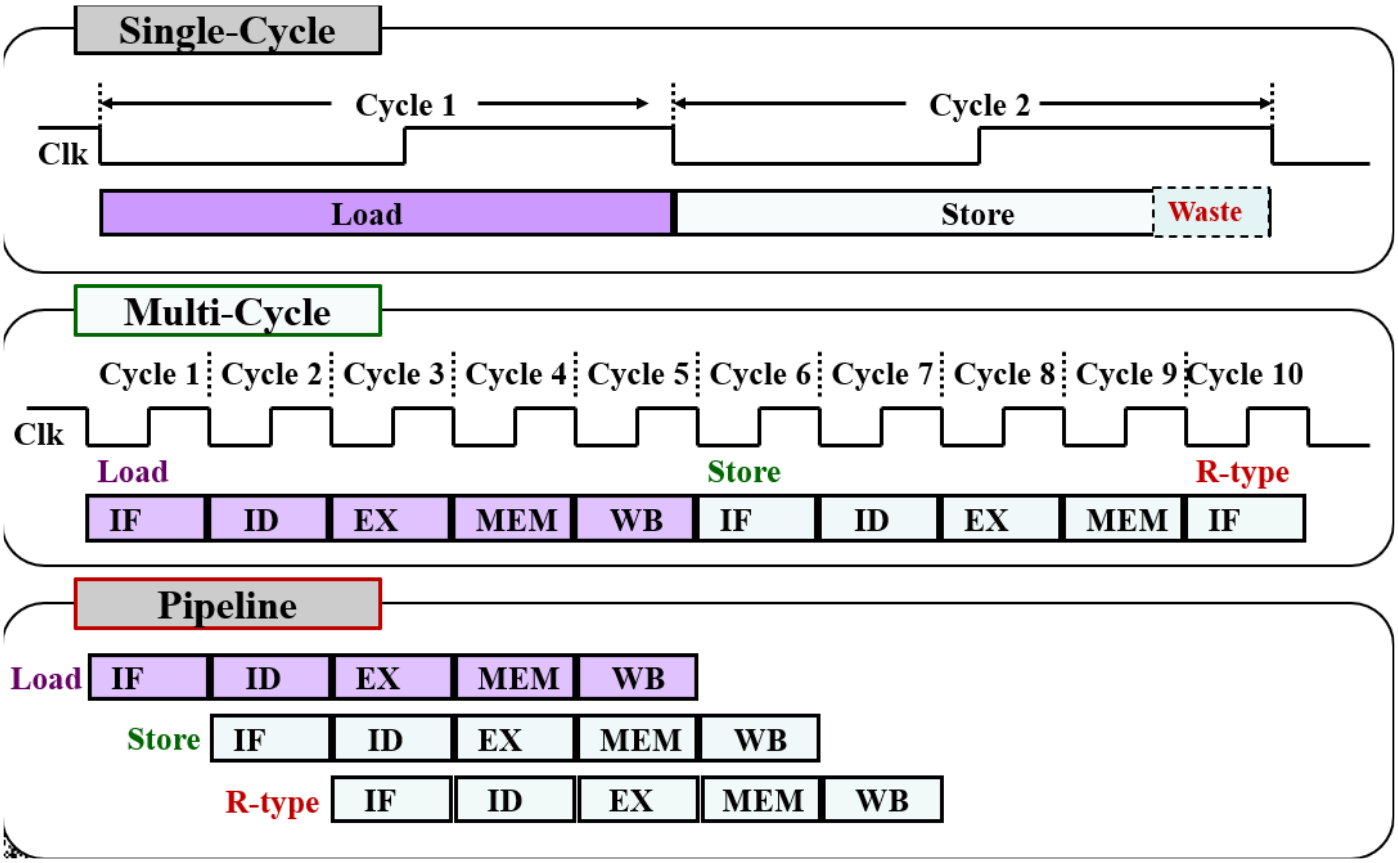


9.4.3 Datapath and Control



9.5 Pipeline Performance

Different Implementations:



9.5.1 Single Cycle Processor

Performance

Cycle time:

- $CT_{seq} = \sum_{k=1}^{N} T_k$
- $T_k =$ Time for operation in stage k
- $N =$ Number of stages

Total Execution Time for I instructions:

- $T_{seq} = \text{Cycles} \times \text{Cycle Time} = I \times CT_{seq} = I \times \sum_{k=1}^{N} T_k$

Example

Instruction	Inst Mem	Reg read	ALU	Data Mem	Reg write	Total
ALU	2	1	2		1	6
lw	2	1	2	2	1	8
sw	2	1	2	2		7
beq	2	1	2			5

Cycle time:

- Choose the longest total time = $8\ ns$
- To execute 100 instructions:

  $100 \times 8\ ns = 800\ ns$

9.5.2 Multi-Cycle Processor

Performance

Cycle time:

- $CT_{multi} = max(T_k)$
- $max(T_k) =$ longest stage duration among the N stages

Total Execution Time for I instructions:

- $T_{multi} =$ Cycles $\times$ Cycle Time $= I \times$ Average CPI $\times CT_{multi}$
- Average CPI is needed because each instruction takes different number of cycles to finish

Example

Instruction	Inst Mem	Reg read	ALU	Data Mem	Reg write	Total
ALU	2	1	2		1	6
lw	2	1	2	2	1	8
sw	2	1	2	2		7
beq	2	1	2			5

Cycle Time:

- Choose the longest stage time = $2\ ns$

To execute 100 instructions, with a given average CPI of 4.6:

- $100 \times 4.6 \times 2\ ns = 920\ ns$

9.5.3 Pipeline Processor

Performance

Cycle Time:

- $CT_{pipeline} = max(T_k) + T_d$
- $max(T_k) =$ longest stage duration among the N stages
- $T_d =$ Overhead for pipelining, e.g. pipeline register

Cycles needed for I instructions:

- $I + N - 1$
- $N - 1$ is the cycles wasted in filling up the pipeline

Total Time needed for I instructions:

- $T_{pipeline} = \text{Cycle} \times CT_{pipeline} = (I + N - 1) \times (max(T_k) + T_d)$

Example

Instruction	Inst Mem	Reg read	ALU	Data Mem	Reg write	Total
ALU	2	1	2		1	6
lw	2	1	2	2	1	8
sw	2	1	2	2		7
beq	2	1	2			5

Cycle Time:

- Assume pipeline register delay of $0.5\ ns$
- Longest stage time + overhead = $2 + 0.5 = 2.5\ ns$

To execute 100 instructions:

- $(100 + 5 - 1) \times 2.5\ ns = 260\ ns$

9.5.4 Ideal Speedup

Assumptions for ideal case:

- Every stage takes the same amount of time: $\sum_{k=1}^{N} T_k = N \times T_k$
- No pipeline overhead $\rightarrow T_d = 0$
- Number of instructions I, is much larger than number of stages N

Note: The above also shows how pipeline processor loses performance

$$
\begin{aligned}
Speedup_{pipeline} &= \frac{T_{seq}}{T_{pipeline}} \\
&= \frac{I \times \sum_{k=1}^{N} T_k}{(I + N - 1) \times (max(T_k) + T_d)} \\
&= \frac{I \times N \times T_k}{(I + N - 1) \times T_k} \\
&\approx \frac{I \times N \times T_k}{I \times T_k} \\
&\approx N
\end{aligned}
$$

Conclusion: Pipeline processor can gain N times speedup, where N is the number of pipeline stages

流水线处理器相对于顺序（非流水线）处理器的性能加速。这里使用了数学公式来阐明这种加速是如何计算的，并指出在什么条件下可以达到理想的加速。我们将逐步解释这个过程。

假设条件：

1. **每个阶段的时间相同：** 这意味着流水线的每个阶段都需要相同的时间来完成其任务，表示为 T_k。
2. **没有流水线开销：** 也就是说，不存在因为指令在流水线中移动而产生的额外时间延迟或者是需要额外的周期。这被表示为 $T_d = 0$（这里的 T_d 是指流水线开销）。
3. **指令数量远大于阶段数：** 这意味着执行的指令数量 I 远大于流水线的阶段数 N，从而任何与流水线启动和结束相关的开销相对于整个执行过程是可以忽略不计的。

加速计算：

首先，我们定义了顺序执行时间 T_{seq} 和流水线执行时间 $T_{pipeline}$，然后我们计算两者的比值以得到加速比 $Speedup_{pipeline}$。

1. **顺序执行时间** 是所有指令在非流水线设置中执行所需的总时间，计算为指令数 I 乘以每条指令的执行时间（由于每个阶段都花费 T_k 时间，总时间为 $N \times T_k$）。

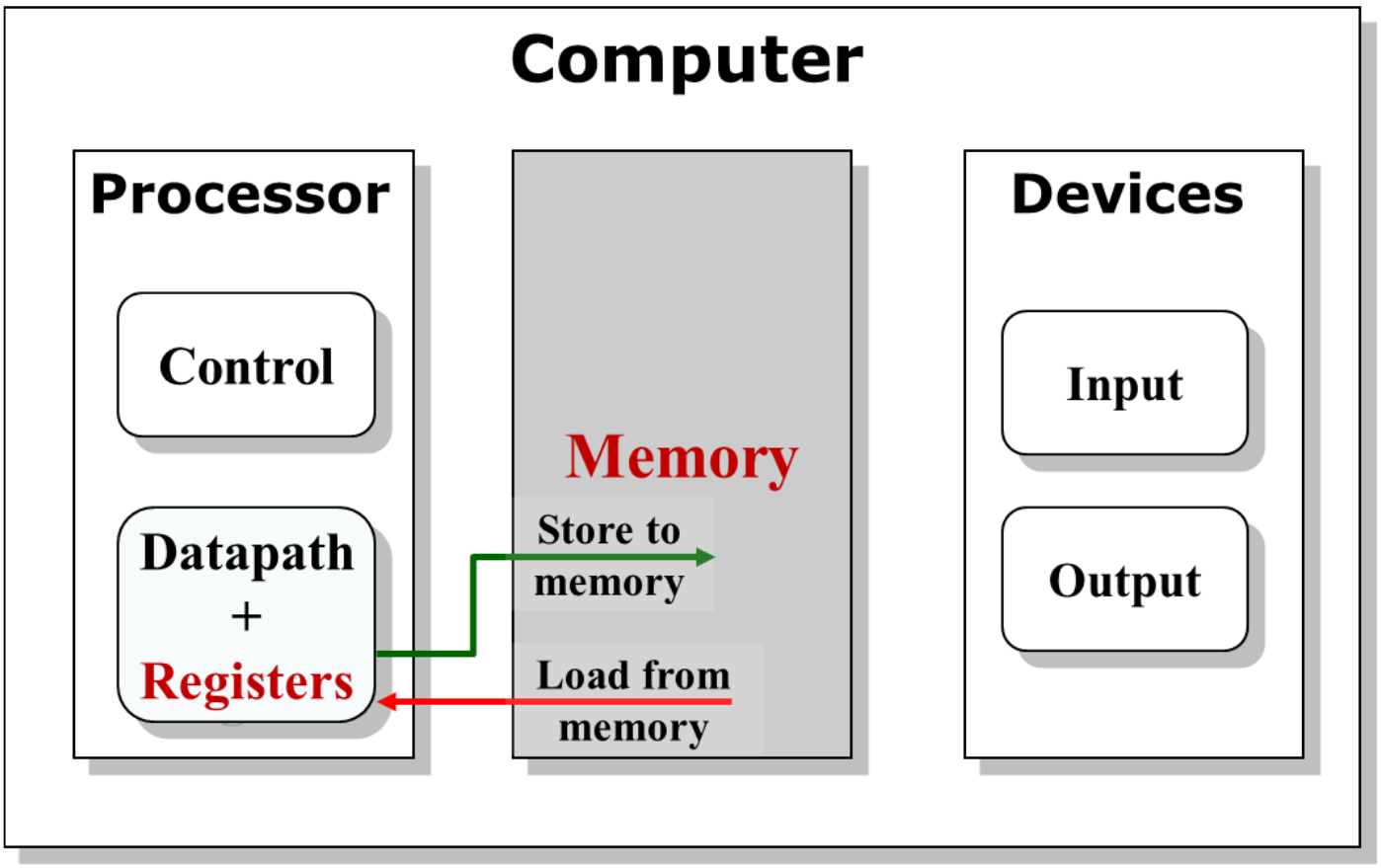
2. **流水线执行时间** 是在流水线设置中执行所有指令所需的时间。重要的是要注意，第一条指令完成之后，每个额外的周期都会完成一条新指令。因此，总时间包括了初始的 N 个阶段和随后的 $I-1$ 个周期（每条指令一个），再加上流水线的开销时间 T_d，但在我们的理想假设中，$T_d = 0$。
3. **加速比** 是顺序时间与流水线时间的比率。在这种情况下，因为每个阶段花费相同的时间并且没有额外的流水线开销，这个比率简化为 $I \times N \times T_k$ 除以 $I \times T_k$（因为 N 阶段花费 $N \times T_k$ 时间，总共有 I 条指令）。当指令数量 I 很大时，$I + N - 1$ 约等于 I，因此加速比接近 N。

结论：

在理想情况下，流水线处理器可以实现 N 倍的加速，其中 N 是流水线的阶段数。这意味着如果您有一个5阶段的流水线，理论上您的处理器速度可以提高5倍。然而，这个理想情况很少出现，因为实际的流水线执行通常会遇到各种开销和延迟，例如由于数据依赖性、控制依赖性、资源冲突等引起的流水线停顿。所以，实际的加速比通常会低于理想情况。

10 - Cache

10.1 Introduction



Registers are in the datapath of the processor. If operands are in memory we have to load them to processor (registers), operate on them, and store them back to memory.

DRAM

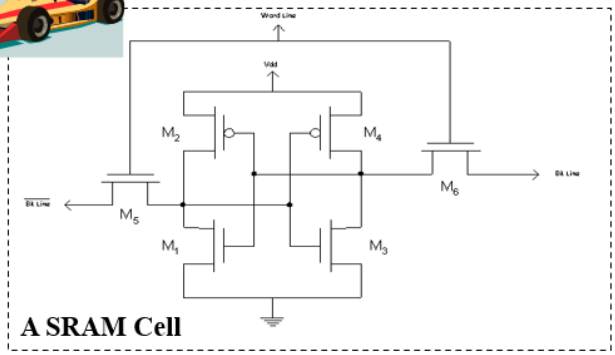
DRAM = DDR SDRAM = Double Data Rate Synchronous Dynamic RAM

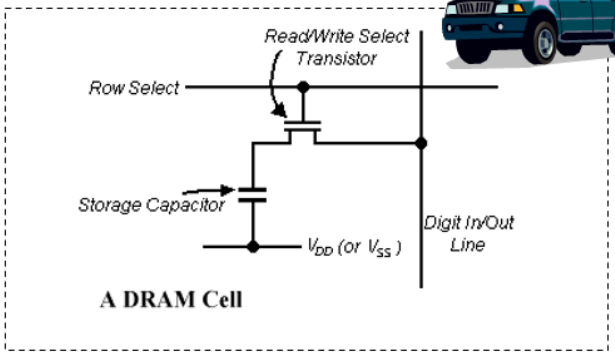
Delivers memory on the positive and negative edge of a clock (double rate)

Generations:

1. DDR (`MemClkFreq x 2 (double rate) x 8 words`)
2. DDR2 (`MemClkFreq x 2 (double rate) x 2 x 8 words`)
3. DDR3 (`MemClkFreq x 4 (double rate) x 2 x 8 words`)
4. DDR4 (Lower power consumption, higher bandwidth)

SRAM





SRAM

6 transistors per memory cell

→ **Low density**

Fast access latency of 0.5 – 5 ns

More costly

Uses flip-flops

DRAM

1 transistor per memory cell

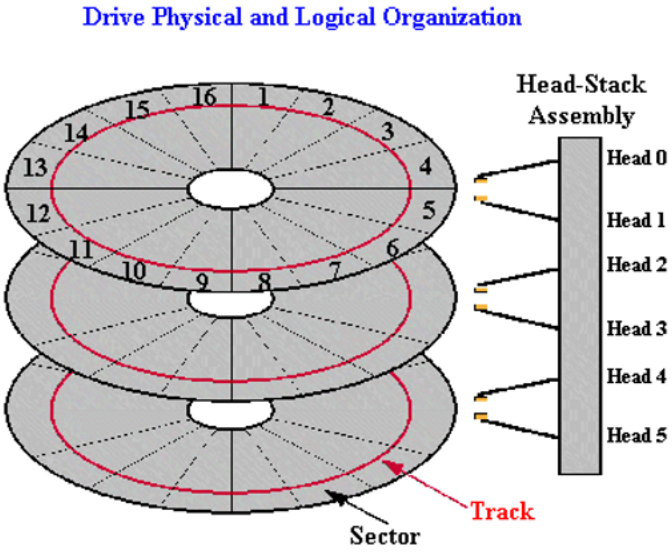
→ **High density**

Slow access latency of 50-70ns

Less costly

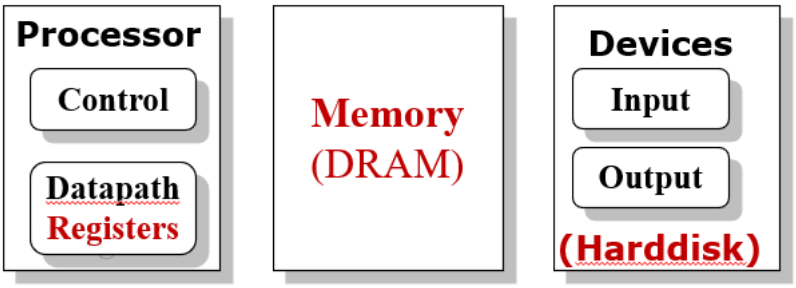
Used in main memory

Magnetic Disk



Typical high-end hard disk:

Average Latency: 4 - 10 ms
Capacity: 500-2000GB



	Capacity	Latency	Cost/GB
Register	100s Bytes	20 ps	$$$$
SRAM	100s KB	0.5-5 ns	$$$
DRAM	100s MB	50-70 ns	$
Hard Disk	100s GB	5-20 ms	Cents
Ideal	**1 GB**	**1 ns**	**Cheap**

10.2 Cache

Basic Idea

Keep the frequently and recently used data in smaller but faster memory

Refer to bigger and slower memory:

- Only when you cannot find data/instruction in the faster memory

Principle of Locality:

- Program accesses only a small portion of the memory address space within a small time interval

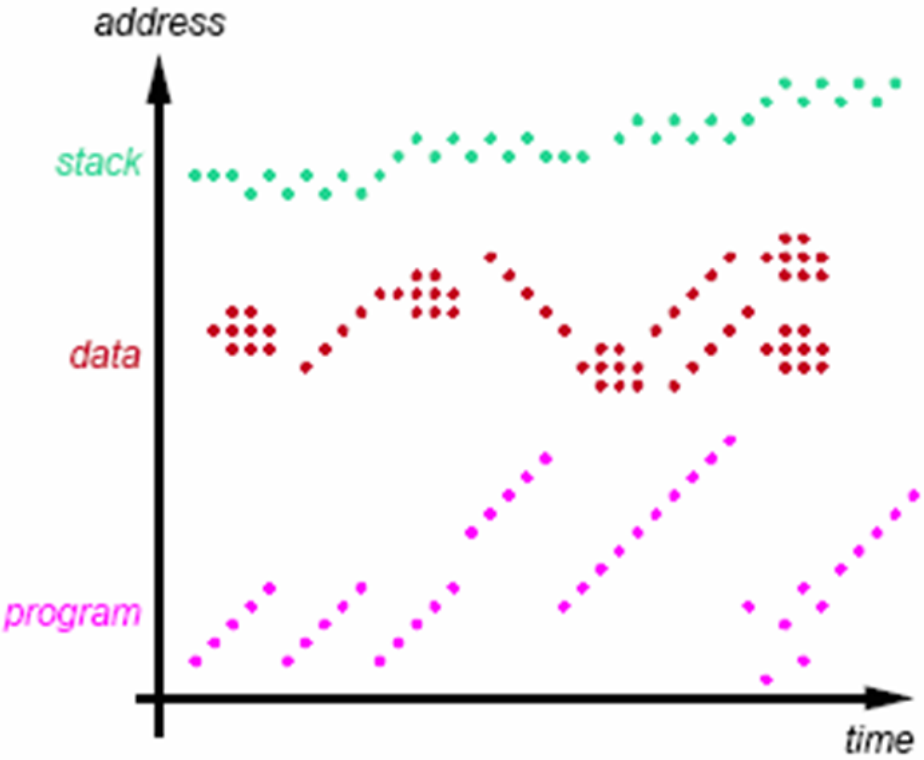
10.2.1 Types of locality

Temporal locality:

- If an item is referenced, it will tend to be referenced again soon

Spatial locality:

- If an item is referenced, the nearby items will tend to be referenced soon



Working Set

Set of locations accessed during Δt

Different phase of execution may use different working sets

Our aim is to capture the working set and keep it in the memory closet to CPU

工作集（Working Set）

工作集是一个动态概念，指的是在一段特定的时间间隔Δt内，程序访问的所有唯一的内存位置（或页面）的集合。这个时间间隔可以是固定的，也可以是基于特定的执行阶段。工作集的大小和内容可能会随着程序执行阶段的不同而变化。

执行阶段和工作集

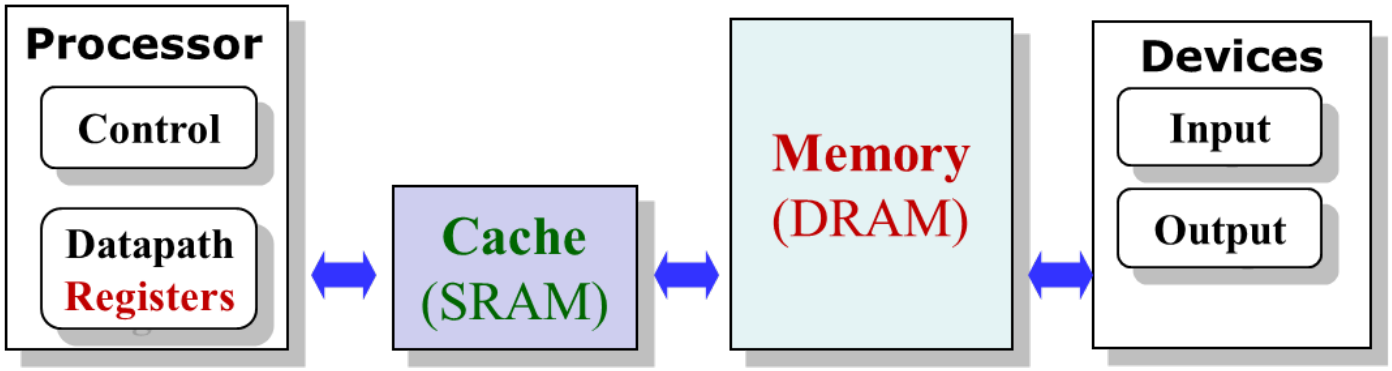
程序在其生命周期中可能会经历多个不同的执行阶段，每个阶段可能会使用不同的数据集和代码路径。例如，在初始化阶段，程序可能会加载某些库和数据结构，而在执行计算或响应用户输入的阶段，则可能访问完全不同的内存区域。这些阶段可能具有不同的工作集。

为什么工作集重要?

1. **性能优化：** 了解程序的工作集对于优化性能至关重要。如果可以将工作集中的数据和代码保留在接近 CPU 的内存（例如，缓存或主内存）中，那么程序的性能可以大幅提高，因为 CPU 访问近处的内存比访问磁盘等远程存储设备要快得多。
2. **内存管理：** 操作系统的内存管理器试图优化可用内存的分配，确保最频繁访问的数据（即工作集）位于最快的存储区域。这通常通过页面替换算法来实现，该算法会根据工作集的变化来调整哪些页面应该留在内存中，哪些应该被换出到辅助存储（如磁盘）。

3. **预测和调度：** 通过监视工作集的变化，操作系统可以预测程序未来的行为和资源需求，从而更智能地进行任务调度和资源分配

10.2.2 Memory Access



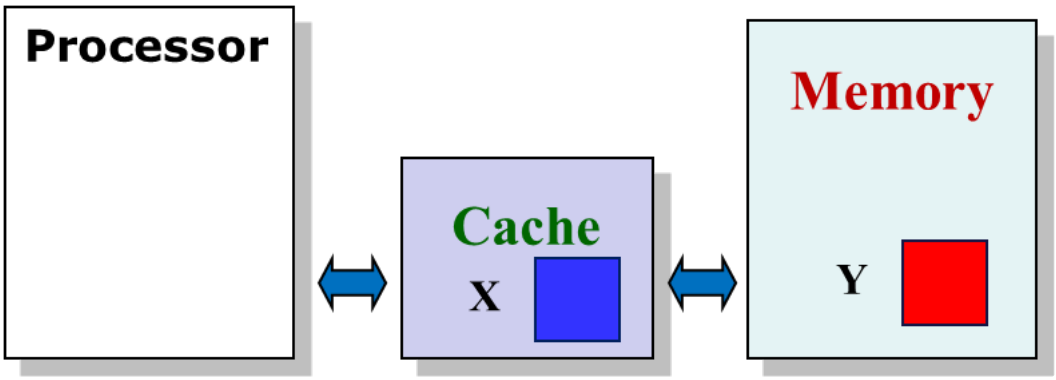
How to make SLOW main memory appear faster?

- Cache - a small but fast SRAM near CPU
- Hardware managed: Transparent to programmer

How to make SMALL main memory appear bigger?

- Virtual memory
- OS managed: Transparent to programmer

Memory Access Time: Terminology



Hit: Data is in cache

- Hit rate: Fraction of memory accesses that hit
- Hit time: Time to access cache

Miss: Data is not in cache

- Miss rate = 1 - Hit rate
- Miss penalty: Time to replace cache block + hit rate

Hit time < Miss penalty

Memory Access Time: Formula

$$\text{Average Access Time} = \text{Hit rate} \times \text{Hit time} + (1 - \text{Hit rate}) \times \text{Miss penalty}$$

Example:

- Suppose our on-chip SRAM (cache) has **0.8 ns** access time, but the fastest DRAM (main memory) we can get has an access time of **10ns. How high a hit rate** do we need to sustain an average access time of **1ns**?
  - Let h be the desired hit rate.

    $1 = 0.8h + (1 - h) \times (10 + 0.8)$
    $= 0.8h + 10.8 - 10.8h$

    $10h = 9.8 \rightarrow h = 0.98$

    Hence we need a hit rate of **98%.**

10.3 Memory to Cache Mapping

Cache Block/Line:

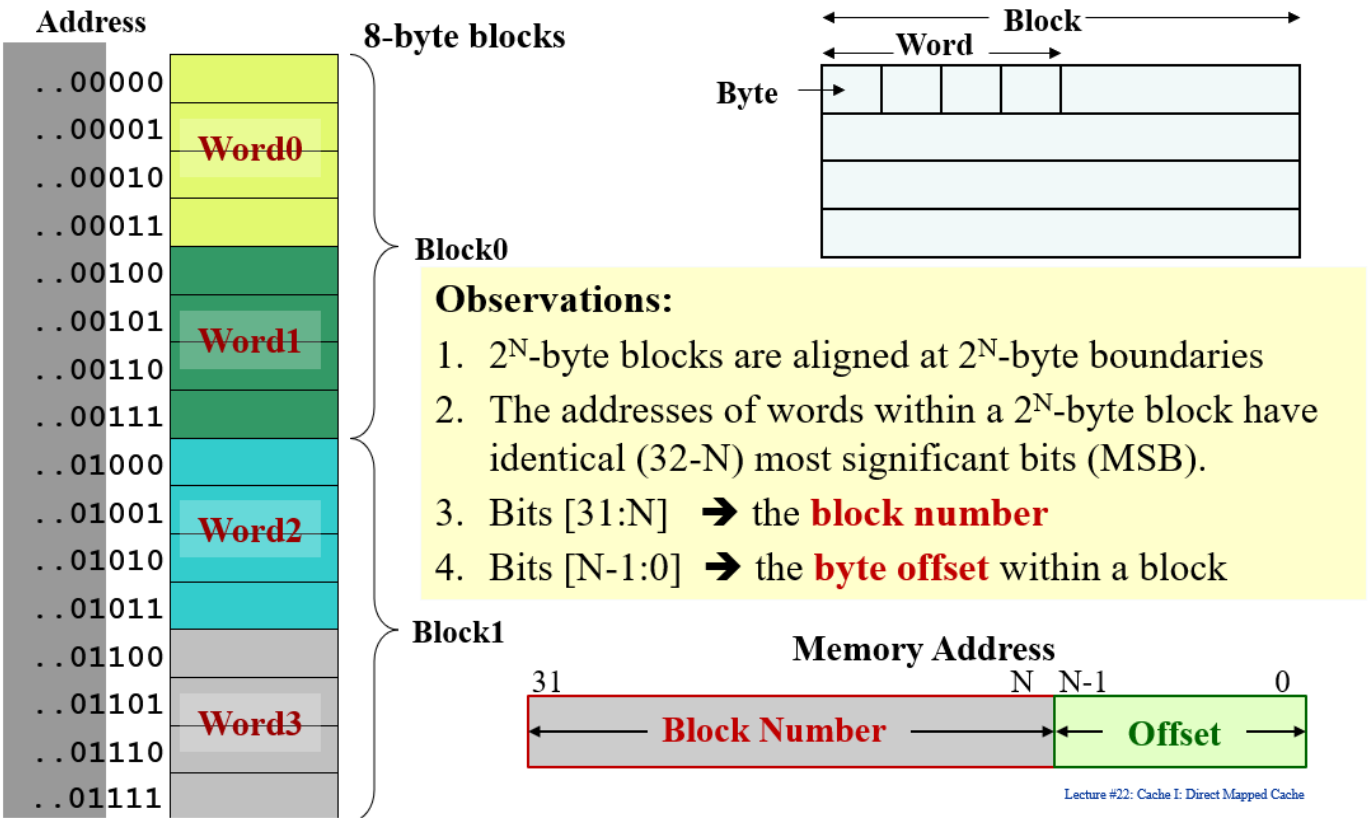
- Unit of transfer between memory and cache

Block size is typically one or more words

- e.g.: 16-byte block ≈ 4-word block
- 32-byte block ≈ 8-word block

Why the block size is bigger than word size?

1. **空间局部性（Spatial Locality）：**
   - 程序倾向于访问最近访问过的内存位置附近的内存位置。这是由于程序的结构通常是按顺序执行的，并且数据也往往是以数据结构（如数组、结构体等）的形式组织的。
   - 因此，当一个特定的内存地址被访问时，其附近的地址也很可能很快被访问。将这些数据作为较大的块（而非单个字）一起存储在缓存中，可以预加载这些可能很快就需要的数据，从而减少了未来的缓存未命中。
2. **时间局部性（Temporal Locality）：**
   - 程序倾向于在短时间内多次访问相同的内存位置。通过在缓存中保存一个比单个字更大的块，系统可以在连续的操作中更有效地使用这些数据，而不是频繁地从主内存中重新加载。
3. **传输效率：**
   - 从主内存到缓存的数据传输是以固定大小的块进行的。传输较大的数据块（而非单个字）更能有效利用内存带宽，因为每次传输都涉及一定的启动（开销）成本。较大的块意味着相对较少的传输，从而减少了总体延迟。
4. **降低缓存未命中率（Cache Misses）：**
   - 当CPU访问的数据不在缓存中时，就会发生缓存未命中。较大的缓存块可以减少缓存未命中的可能性，因为更多的相关数据已经预加载到缓存中。



10.4 Direct Mapped Cache

直接映射缓存的工作原理：

在直接映射缓存中，主存储器（main memory）被划分为多个块（blocks），而缓存（cache）则被划分为多个线（lines）或槽（slots）。这些缓存线用于存储来自主存储器的数据块。当CPU需要读取特定地址的数据时，系统会检查这些数据是否已在缓存中。

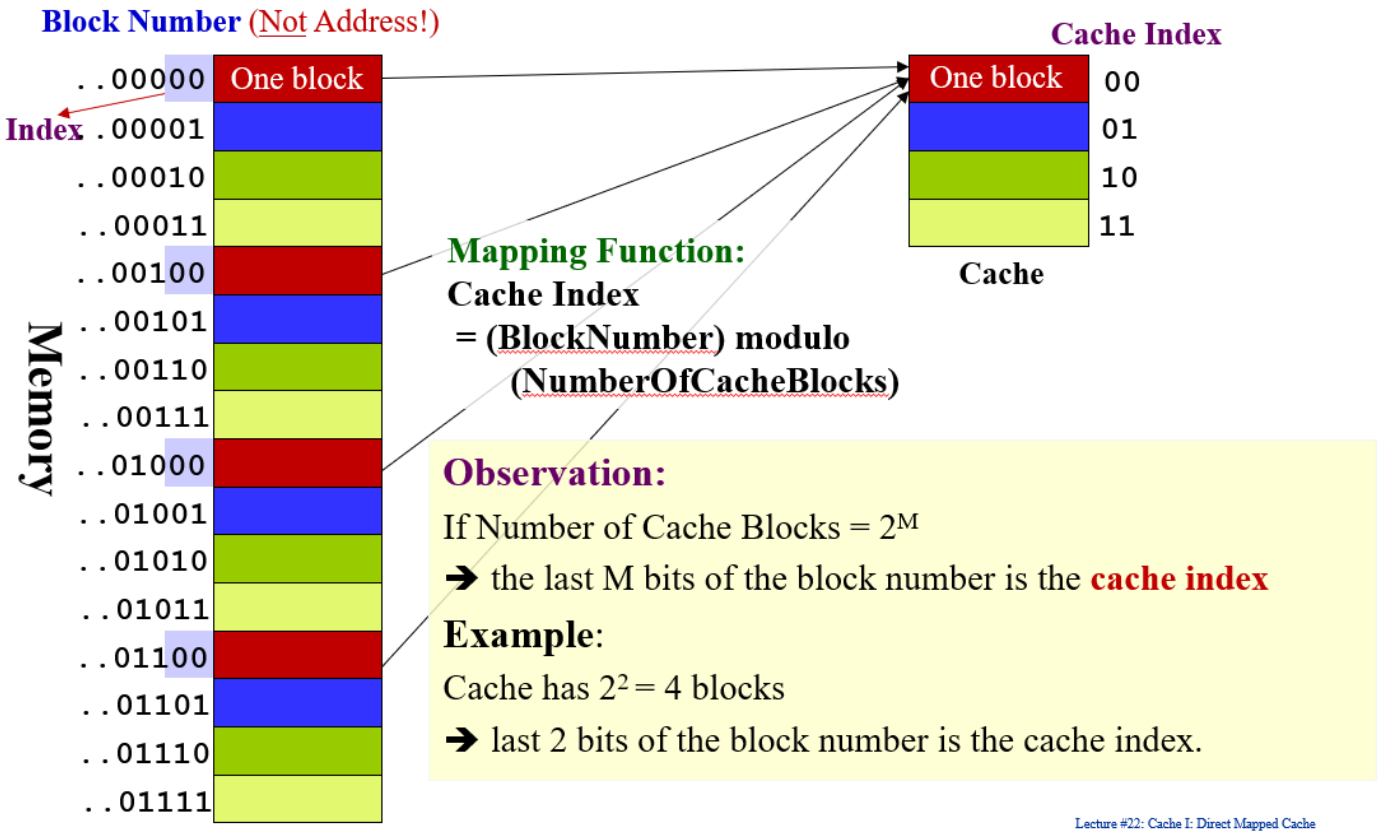
为了决定一个内存块应该存储在缓存的哪个位置，以及如何在缓存中找到这些块，系统会使用一种映射策略。在直接映射缓存中，使用了内存地址的一部分来确定一个特定的缓存线。

缓存索引的作用：

内存地址通常包含以下几个部分：

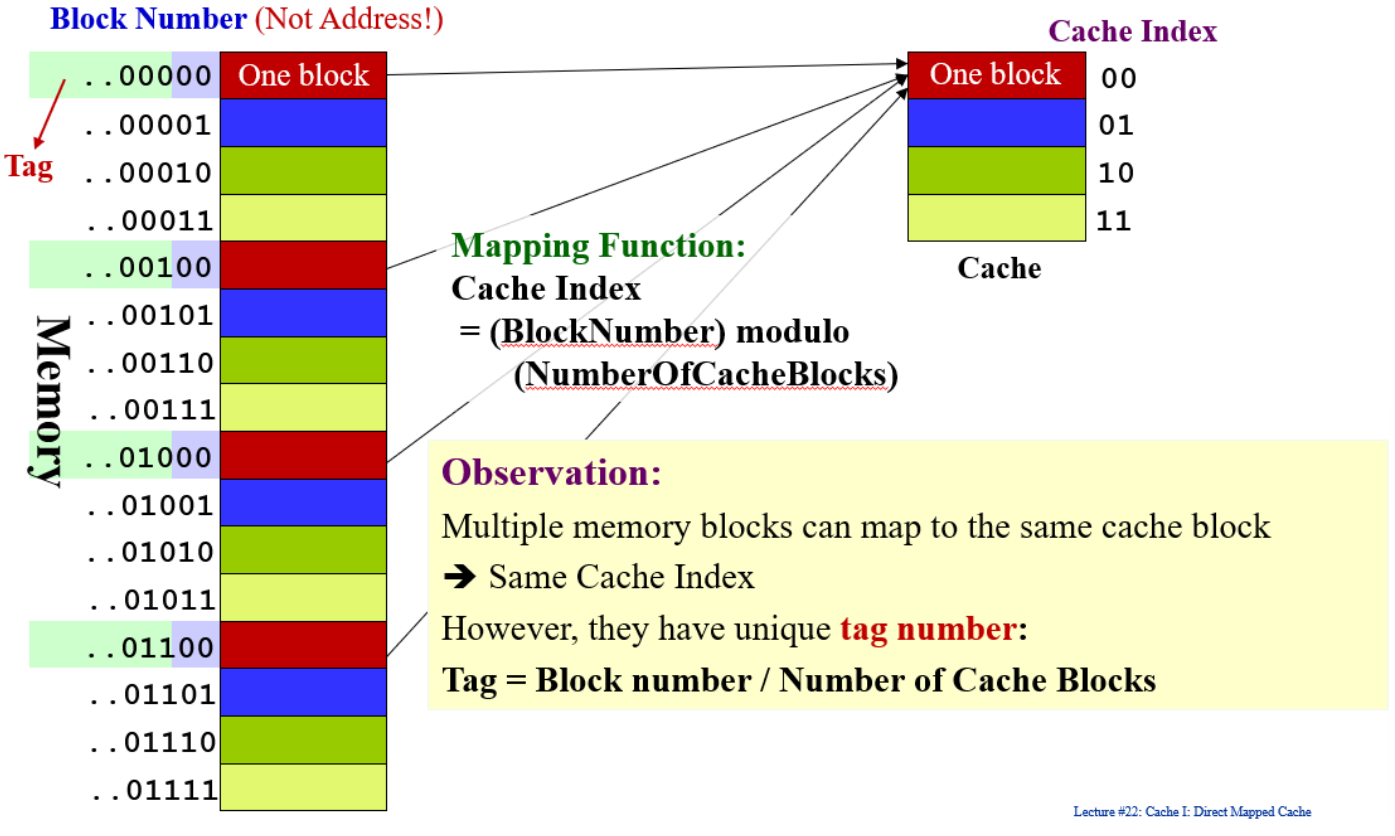
1. **标记（Tag）**：用于唯一标识一个内存块。当缓存检查某个特定缓存线时，它会比较地址标记和缓存线中存储的标记，以确定所需数据是否在该缓存线中。
2. **索引（Index）**：这是直接决定内存块在缓存中位置的部分。系统通过内存地址的索引字段来选择应该使用哪个缓存线。换句话说，索引用于“直接映射”内存块到特定的缓存线。
3. **块内偏移（Block Offset）**：当存储块大小大于一个字（word）时，块内偏移用于在一个块内部定位特定的字。

10.4.1 Cache Index and Tag



计算索引：

- 即我们需要最少的比特位数来表示所有的block。如果一共有2^M个block，那么我们需要M个bit位才能表示所有的block，所以block number的最后M位即位索引位



标记(Tag)

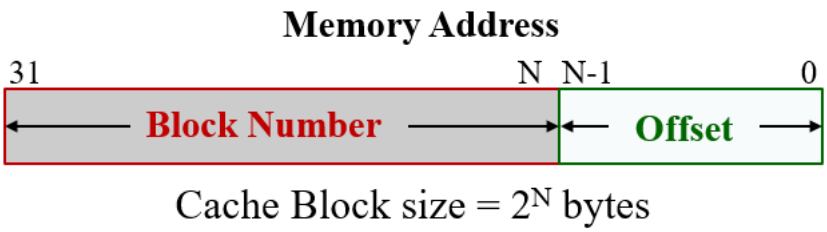
定义与功能：

- "标记"是内存地址的一部分，用于在缓存中唯一标识一个数据块。由于缓存容量比整个内存小，因此不可能将所有内存块同时加载到缓存中。标记用于区分不同的内存块，即使它们可能映射到缓存的同一索引位置。
- 当处理器试图访问缓存时，系统会检查所选缓存行中存储的标记与当前请求的内存地址的标记部分是否匹配。如果这两个标记相同，就会发生“缓存命中”；否则，就会发生“缓存未命中”。

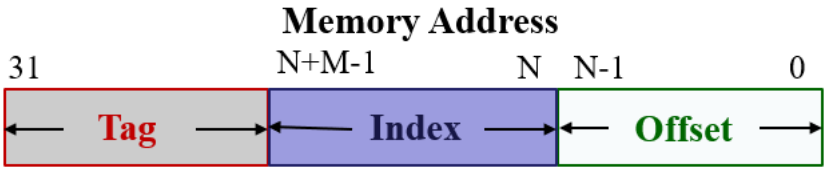
计算标记：

- `Tag = Block number / Number of Cache Blocks`

10.4.2 Mapping



Cache Block size = 2^N bytes



Cache Block size = 2^N bytes

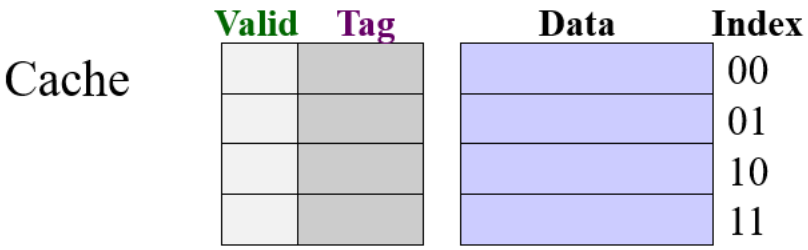
Number of cache blocks = 2^M

Offset = N bits

Index = M bits

Tag = 32 – (N + M) bits

10.4.3 Cache Structure



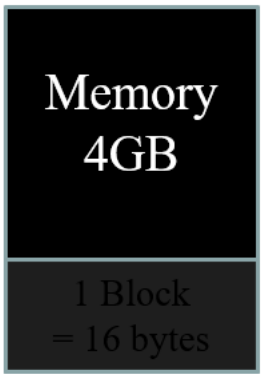
Along with a data block(line), cache also contains the following administrative information

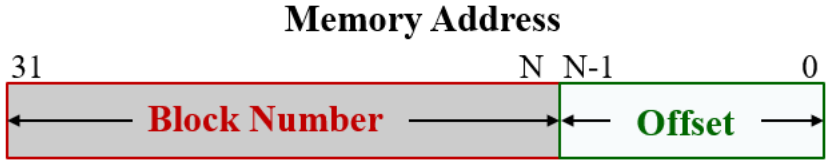
1. Tag of the memory block
2. Valid bit indicating whether the cache line contains valid data

When is there a cache hit?

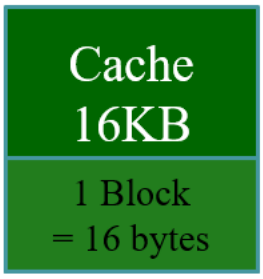
`Valid[index] == True AND Tag[index] == Tag[memory address]`

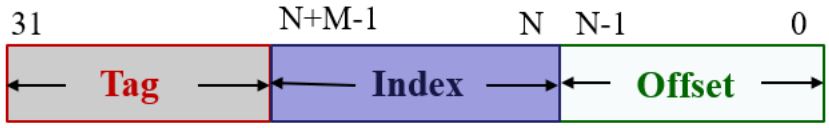
10.4.4 Example





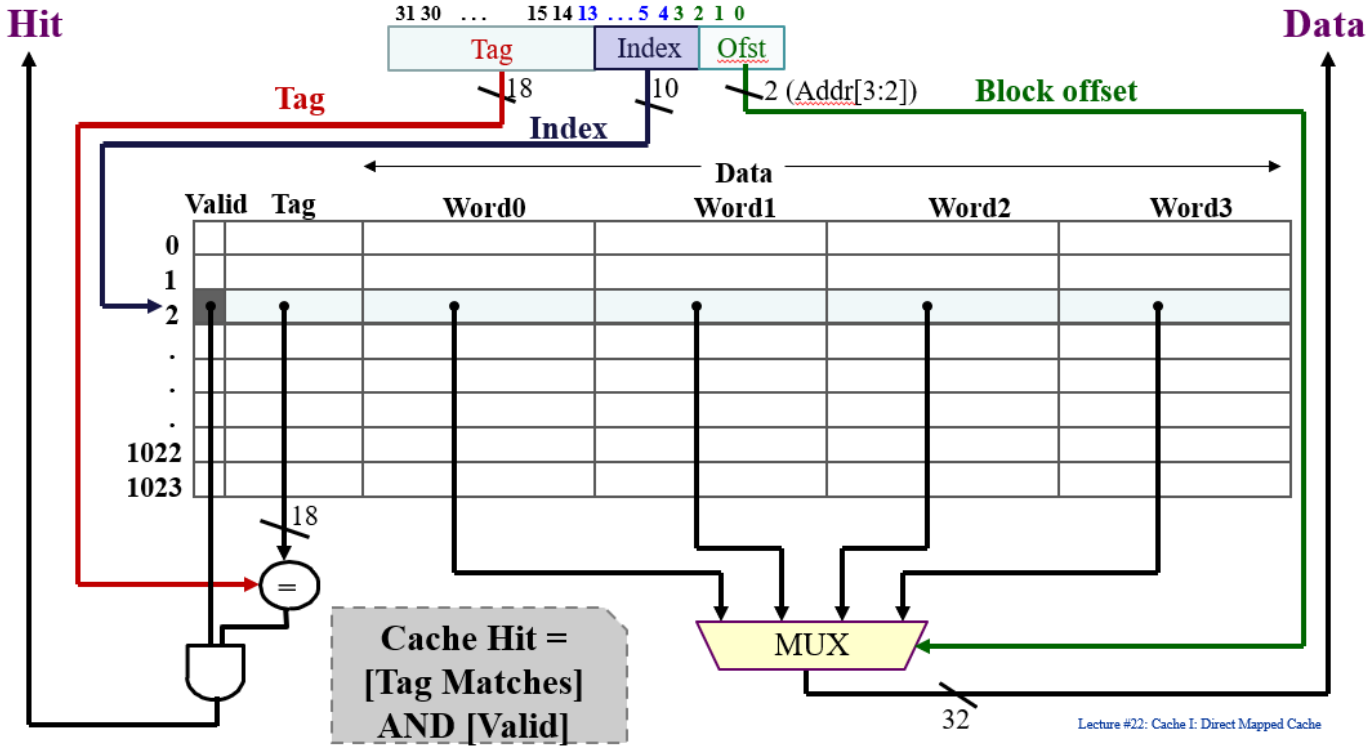
Offset, N = 4 bits
Block Number = 32 – 4 = **28 bits**
Check: Number of Blocks = 2^{28}





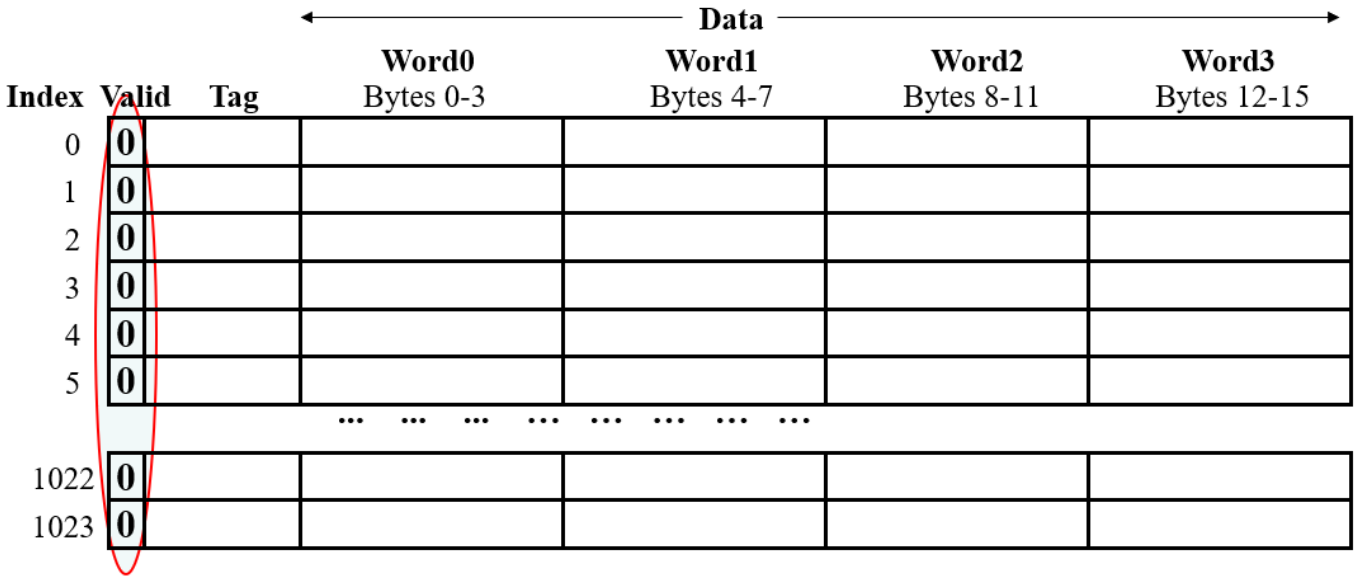
Number of Cache Blocks
= 16KB / 16bytes = 1024 = 2^{10}
Cache Index, M = 10bits
Cache Tag = 32 – 10 – 4 = **18 bits**

16-KB cache:
4-word (16-byte) blocks



10.5 Reading Data

#1 Initial State



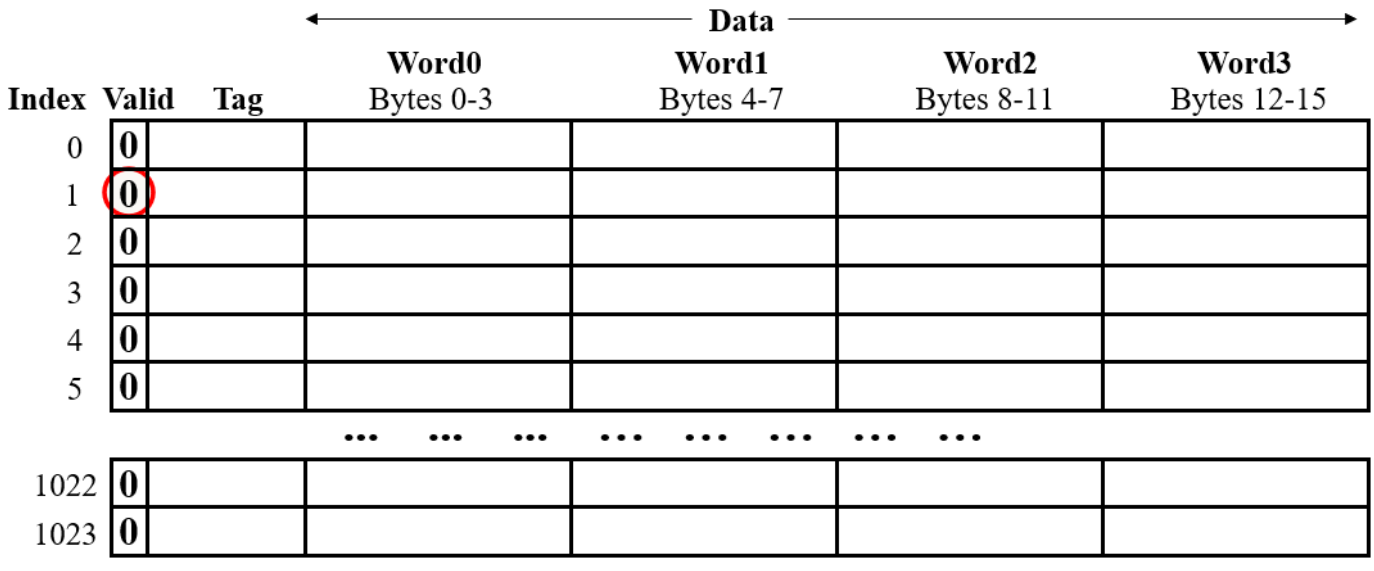
起始状态：

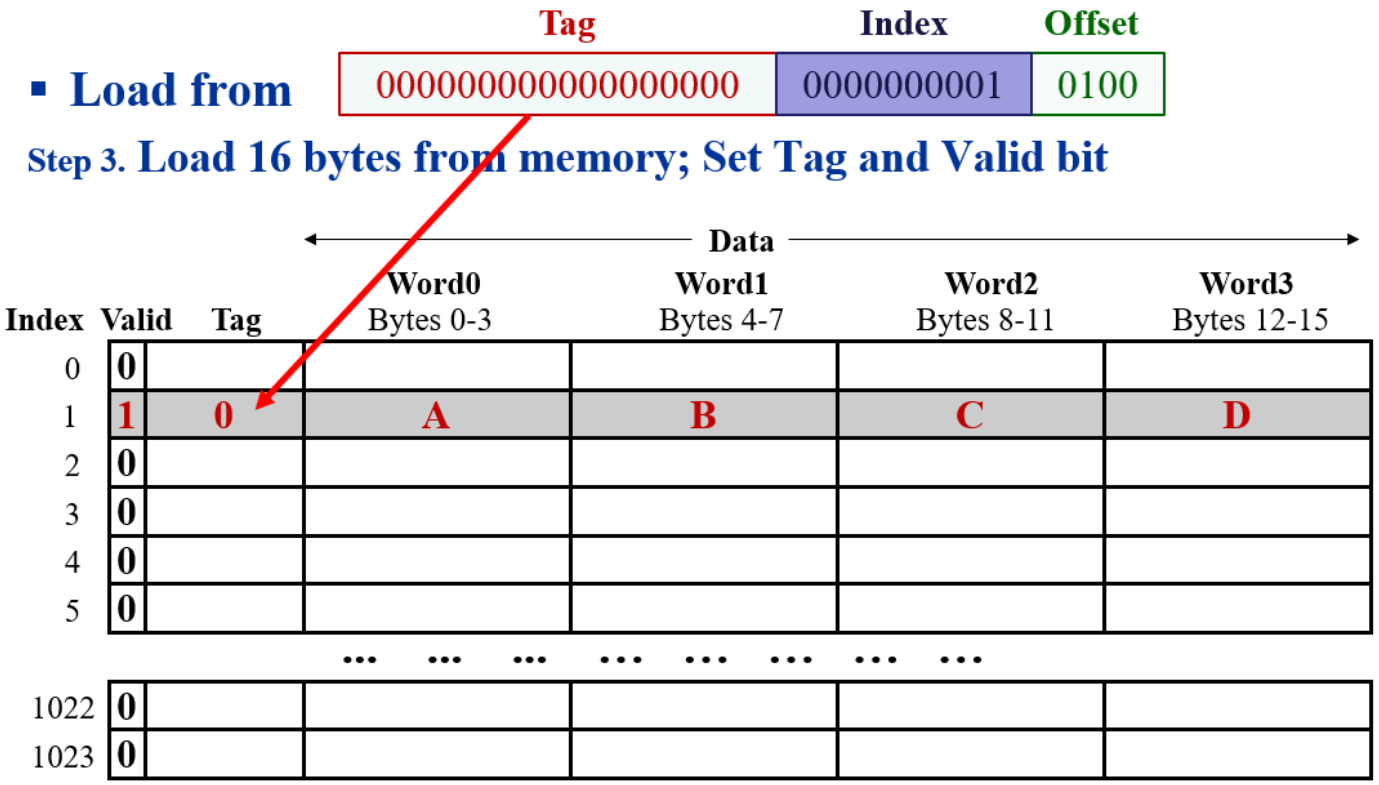
1. 所有的field都是空的
2. 所有的valid bit都为0

> valid bit代表这个slot有没有被写入过数据。在判断缓存命中时需要判断valid bit为1才能正确命中

#2 First read data

Step 2. Data in block 1 is invalid [Cold/Compulsory Miss]

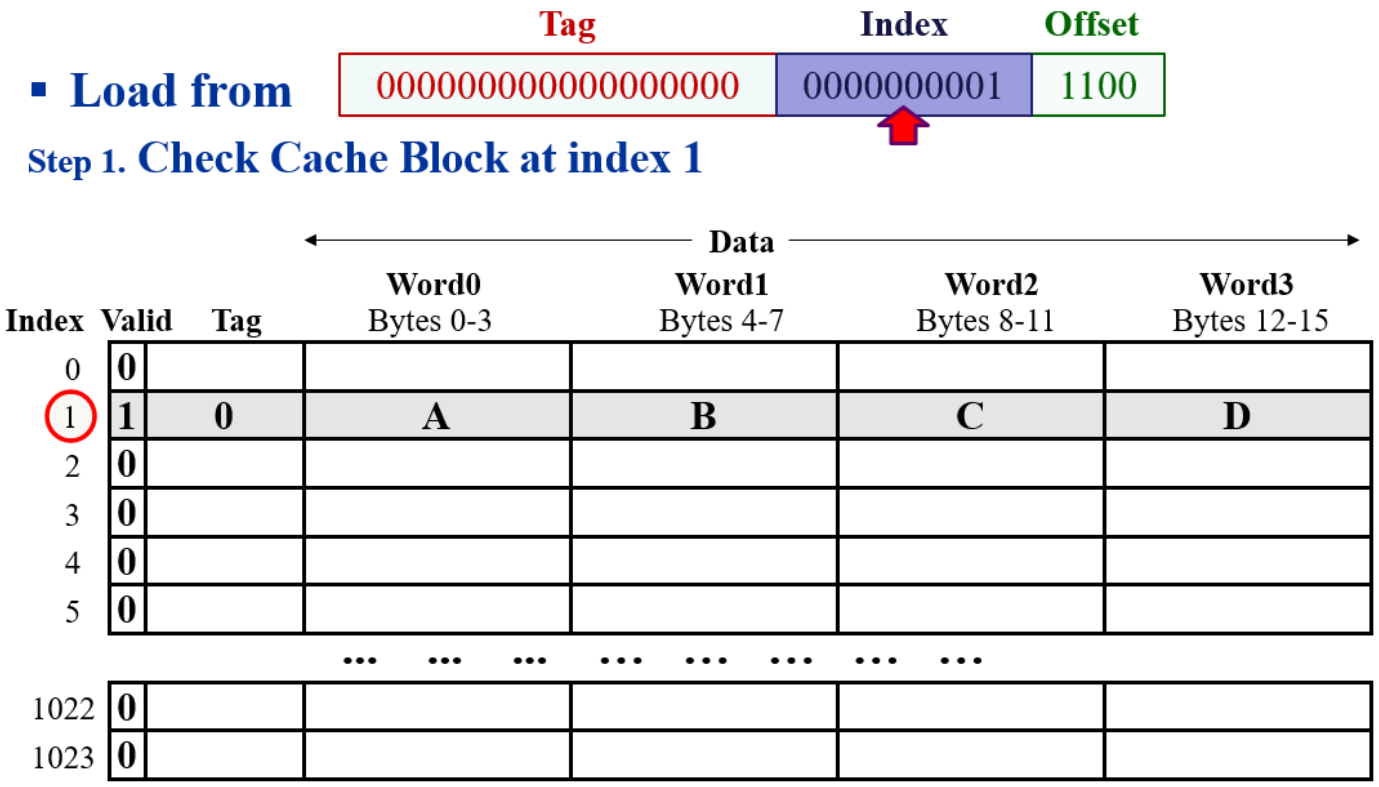




此处读取了index为1，tag为0，offset为0100的块。

1. 首先检索到缓存索引为1的插槽，其valid bit为0，意味着这个slot没有被写入过。判定为cold/compulsory miss
2. 写入tag，和16 bytes (4 words)的数据。虽然寄存器每次只需要1 word的数据，但是根据时间局部性(Temporal locality)和空间局部性(Spatial locality)，我们写入此word所在的整个block。并将valid bit设置为1
3. 读取offset代表的word，0100即byte 4开始的word，即为word1，传递给寄存器

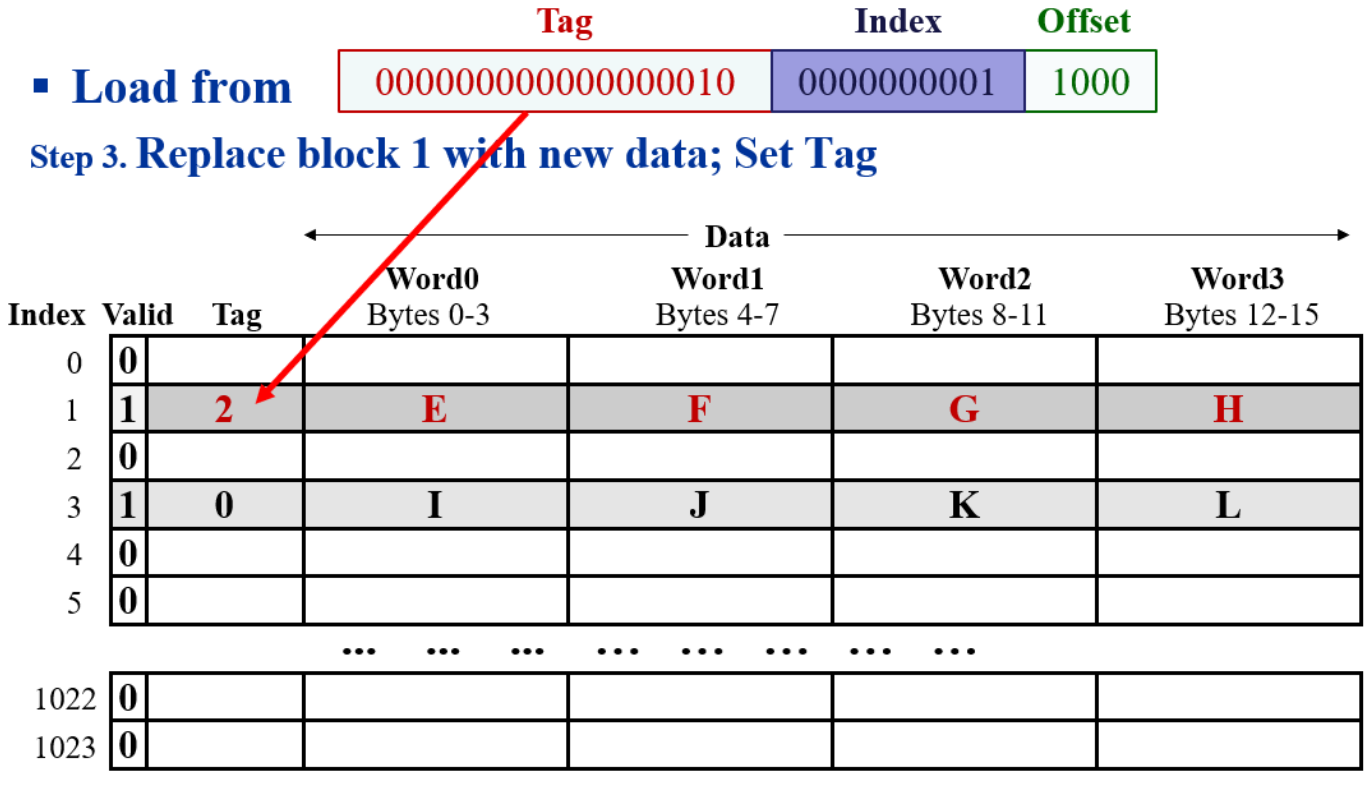
#3 - Cache hit



当寄存器再次需要同一个index但不同offset的word时：

1. 首先检查index索引值，找到所对应的slot
2. 检查valid bit是否为1，以及tag值是否一致。tag作为内存中每个block的特征码是独一无二的
3. 根据offset偏移量确定所需要的word是从1100=12 byte开始的，所以传递word3给CPU寄存器

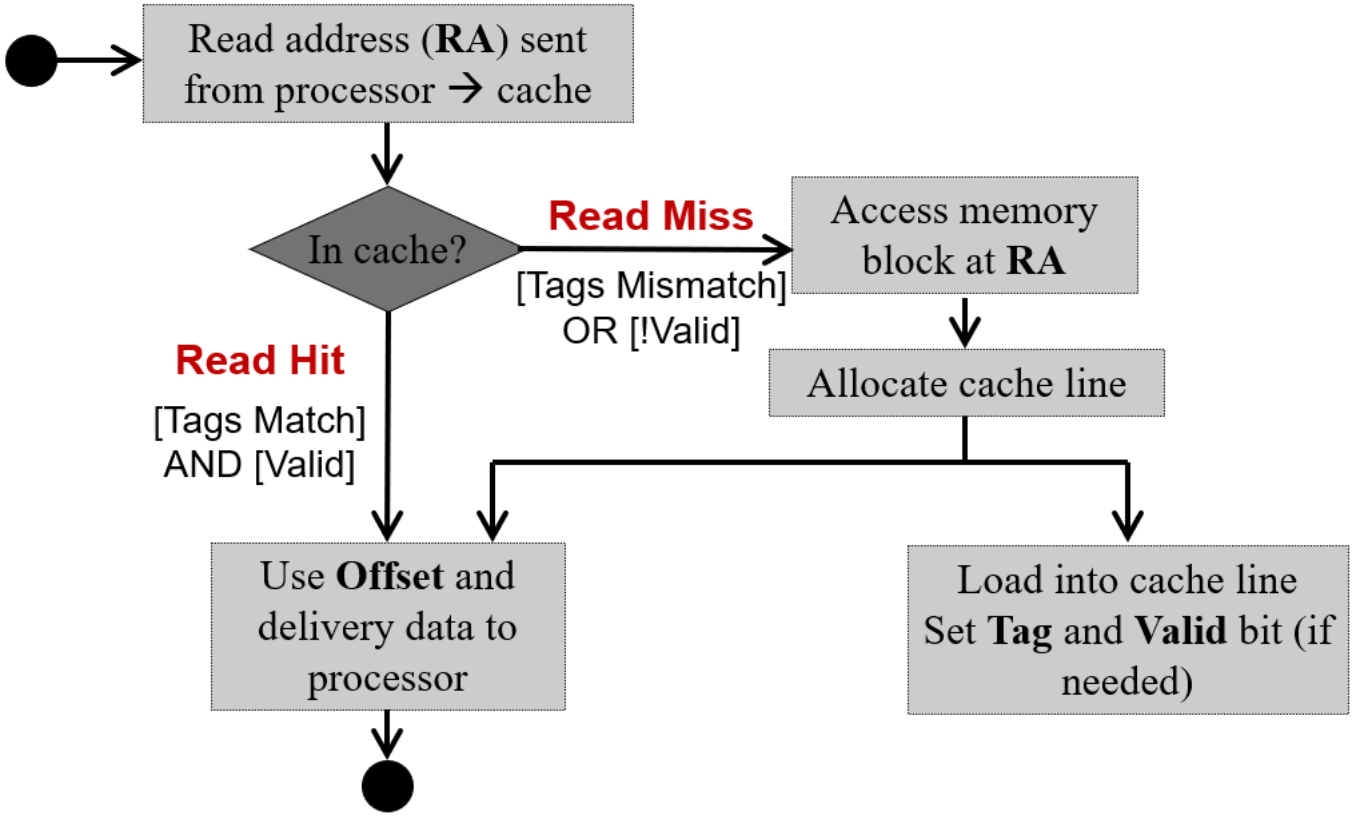
#4 - Different tag



如果index和valid bit能够对应上，但是tag对应不上，依旧被判定为miss（cold miss）

需要替换该index对应的slot中的所有数据，包含tag和data

Summary



10.6 Type of Cache miss

Compulsory misses:

- On the first access to a block; the block must be brought into cache
- Also called cold start misses or first reference misses

Conflict misses:

- Occur in the case of direct mapped cache or set associative cache, when several blocks are mapped to the same block/set
- Also called collision misses or interference misses

Capacity misses

- Occur when blocks are discarded from cache as cache cannot contain all blocks needed

1. **Compulsory Misses（强制性缺失）：**
    - 这类缺失发生在对一个数据块的首次访问；因为数据块还没有被加载到缓存中，所以必须从更低一级的内存（如主内存）中取出数据块。
    - 这也被称为冷启动缺失（cold start misses）或首次引用缺失（first reference misses），因为它们通常发生在程序刚开始运行时，此时缓存未被充分利用。

2. **Conflict Misses（冲突缺失）：**
    - 这类缺失发生在直接映射缓存（direct-mapped cache）或组相联缓存（set-associative cache）中，当多个数据块映射到同一个缓存块或集合（set）时。如果新来的数据块与已经在缓存位置的数据块发生冲突，已在缓存中的数据块将被替换出去。
    - 这也被称为碰撞缺失（collision misses）或干扰缺失（interference misses），因为它们是由于不同数据块之间的映射冲突导致的。
3. **Capacity Misses（容量缺失）：**
    - 这类缺失发生在缓存无法容纳所有需要的数据块时。如果程序访问的数据块数量超过了缓存的容量，缓存中的一些数据块将被替换出去，以便为新的数据块腾出空间。当再次需要被替换出去的数据块时，就会发生容量缺失。
    - 这种情况表明，即使缓存中没有冲突，由于缓存容量本身的限制，也无法避免缺失的发生。

10.7 Changing Cache Content: Write Policy

Cache and main memory are inconsistent

- Modified data only in cache, not in memory

Solution 1: Write-through cache

- Write data both to cache and to main memory

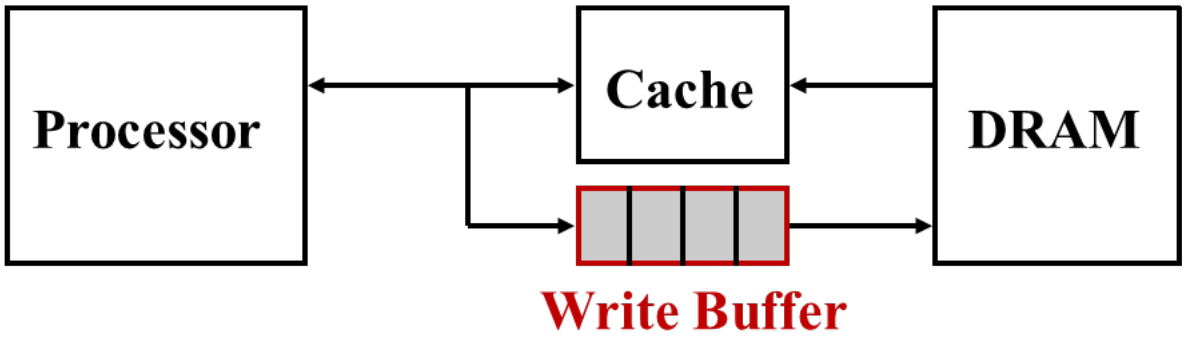
Solution 2: Write-back cache

- Only write to cache
- Write to main memory only when cache block is replaced

当数据被修改后，缓存中的信息和主内存中的信息可能会不一致。为了处理这种不一致性，有两种常见的策略：

1. **写直达（Write-through）缓存：**
    - 在这种策略中，当缓存中的数据被修改时，同时也会将修改的数据写入到主内存中。这样可以确保主内存中的数据始终是最新的，也就是说，缓存和主内存中的数据始终是一致的。
    - 优点是简单、一致性好，当缓存中的数据发生更改时，不需要额外的操作就能保证与主内存的同步。
    - 缺点是每次写操作都需要访问主内存，这会带来额外的时间开销，尤其是当写操作非常频繁时。
2. **写回（Write-back）缓存：**
    - 在这种策略中，修改后的数据仅仅被写回到缓存中，而不是立即写入主内存。只有当缓存中的数据需要被替换，也就是说，当新的数据需要加载到缓存中而缓存已满时，缓存中被修改过的数据（脏数据）才会被写回主内存。
    - 优点是减少了对主内存的写入操作，因为不是每次缓存数据更新都要写入主内存，这可以提高系统的整体性能，特别是在写操作频繁的场景下。
    - 缺点是一致性问题更加复杂。在多核或多处理器系统中，如果不同的处理器缓存中存储了同一主内存位置的不同副本，就需要更复杂的一致性协议来避免数据不一致的问题。

10.7.1 Write-Through Cache



Problem:

- Write will operate at the speed of main memory

Solution:

- Put a write buffer between cache and main memory
  - Processor: writes data to cache + write buffer
  - Memory controller: write contents of the buffer to memory

问题：

- 使用写直达策略时，每次写操作都会同时更新缓存和主存（main memory）。由于主存的写速度通常远低于缓存和处理器的速度，因此每次写操作都会被主存的慢速度拖慢，这影响了整个系统的性能。

解决方案：

- 为了解决这个问题，系统中引入了一个“写缓冲区”（write buffer）。写缓冲区是一种快速存储器，位于缓存和主存之间。
  - 当处理器执行写操作时，它只需要将数据写入缓存和写缓冲区。由于这两者的速度都很快，处理器不会因为等待写操作完成而闲置，从而可以继续执行后续操作。
  - 同时，内存控制器（memory controller）会在后台处理写缓冲区中的数据，将其写入较慢的主存中。这一步是异步进行的，不会影响处理器的正常工作。

10.7.2 Write-Back Cache

Problem:

- Quite wasteful if we write back every evicted cache blocks

Solution:

- Add an additional bit (dirty bit) to each cache block
- Write operation will change dirty bit to 1
  - Only cache block is updated, no write to memory
- When a cache block is replaced, only writes back to memory if dirty bit is 1

问题：

- 如果我们在替换缓存块时每次都将其写回主存，这将非常浪费资源，因为不是所有被替换的缓存块都包含更新过的数据。在一些情况下，缓存块的数据可能自从被加载到缓存以来并没有被修改过，因此将其写回主存是不必要的。

解决方案：

- 为了更有效地管理缓存内容的写回，系统引入了“脏位”（dirty bit）这一概念。脏位是附加在每个缓存块上的一个额外的位。
  - 当处理器写入缓存时，它只更新缓存块中的数据，并将对应的脏位设置为1，表明该缓存块已被修改，与主存中的相应块内容不一致。这个过程中，并不会有数据写回到主存。
  - 当需要替换缓存块时，系统会检查脏位。如果脏位为1（表示缓存块已被修改），系统才将该缓存块的内容写回主存。如果脏位为0（表示缓存块自加载后未被修改），则无需进行写回操作。

10.7.3 Handling Cache Misses

On a Read Miss:

- Data loaded into cache and then load from there to register

Write miss option 1: Write allocate

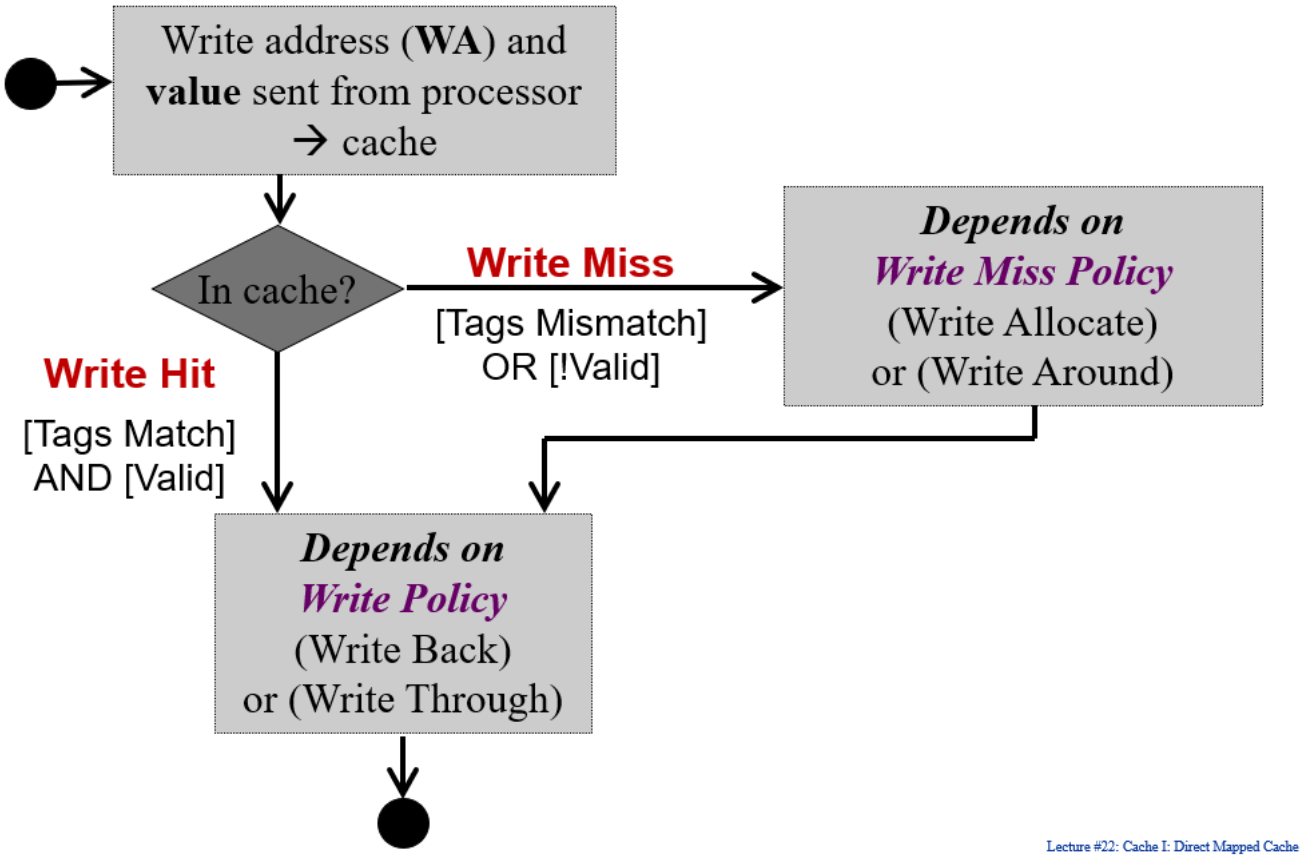
- Load the complete block into cache
- Change only the required word in cache
- Write to main memory depends on write policy (Write-through or Write-back)

Write miss option 2: Write around

- Do not load the block to cache
- Write directly to main memory only

1. **读缺失（Read Miss）：**
   - 当处理器需要读取的数据不在缓存中时（即缓存未命中或读缺失），系统从主存中加载该数据块到缓存中。然后，数据从缓存传输到需要它的处理器寄存器中。这种方式确保了该数据的后续访问能够更快，因为它现在已经在缓存中了。
2. **写缺失（Write Miss）：** 写缺失发生时，处理器想要写的数据不在缓存中。这时有两种主要的处理策略：
   - **写分配（Write Allocate）：**
     - 当写操作缺失时，首先从主存中将整个数据块加载到缓存中。
     - 接着，处理器只更改缓存中相应的必要数据（单个字或字节）。
     - 至于这个更改过的数据是否立即写回到主存，取决于采用的写策略（写直达或写回）。如果是写直达缓存，数据同时会被写到缓存和主存中。如果是写回缓存，数据更新只发生在缓存中，只有在数据块被替换出缓存时，更改才会写回主存。
   - **绕写（Write Around）：**
     - 在这种策略中，当写缺失发生时，数据不会被加载到缓存中。
     - 相反，更改的数据直接写入到主存中，缓存不参与这次操作。
     - 这意味着对这部分数据的后续读取可能会导致缓存缺失，因为数据没有被缓存。这个策略通常用于那些认为不太可能再次用到的数据，或者写操作非常频繁，缓存可能很快就会被新数据填满的情况。

10.7.4 Summary



10.8 Set Associative (SA) Cache

N-way Set Associative Cache

- A memory block can be placed in a fixed number (N) of locations in the cache, where $N > 1$

Key Idea:

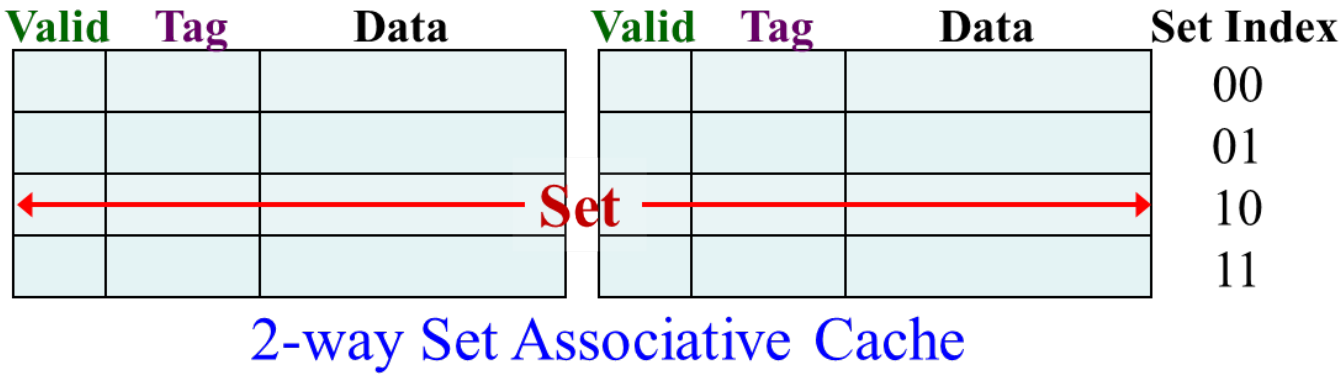
- Cache consists of a number of sets:
  - Each set contains N cache blocks
- Each memory block maps to a unique cache set
- Within the set, a memory block can be placed in any of the N cache block in the set

> Set Associative (SA) Cache是一种折中的缓存映射策略，结合了直接映射缓存（Direct Mapped Cache）的高效性和全关联缓存（Fully Associative Cache）的灵活性。它试图在这两种策略的优势之间找到平衡，减少缓存未命中的次数，同时保持合理的硬件复杂度和成本。
>
> Set Associative缓存的工作原理如下：
>
> 1. **集合（Set）：** 缓存被划分为多个集合（sets），每个集合包含几个缓存行（cache lines）或块（blocks）。这些块是缓存中可以存储数据的单位。
> 2. **关联度（Associativity）：** 每个集合中的块数定义了缓存的“关联度”。例如，如果每个集合有四个块，则该缓存是4路组相联的。

3. **映射：** 当主存中的一个块需要被加载到缓存中时，首先会根据该块的地址计算出它应该存储在哪个集合中。这个计算过程通常基于地址的某些位，并且每个集合对应主存中的多个块。然而，一个给定的块只能映射到一个特定的集合。
4. **替换策略（Replacement Policy）：** 如果计算出的目标集合已满（即每个块都已被其他数据占用），缓存必须决定哪个块将被替换以腾出空间。这是通过所谓的替换策略来完成的，常见的替换策略有最近最少使用（LRU）、随机（Random）等。

通过这种方式，Set Associative缓存降低了发生冲突缺失（多个内存地址映射到同一缓存位置）的可能性，因为现在每个内存块有多个潜在的缓存位置可供选择。这增加了一些查找所需数据的复杂性（因为现在必须在一个集合的所有块中查找），但通常能够显著提高缓存的命中率，尤其是在工作集大小适中，且访问模式较为分散的情况下。

10.8.1 Structure



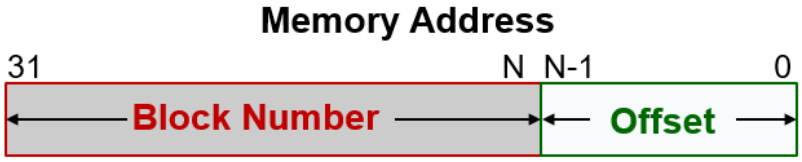
An example of 2-way set associative cache

- Each set has two cache blocks

A memory block maps to a unique set

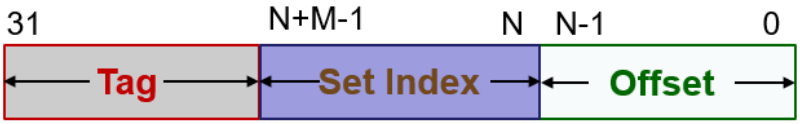
- In the set, the memory block can be placed in either of the cache blocks
- Need to search both to look for the memory block

10.8.2 Mapping



Cache Block size = 2^N bytes

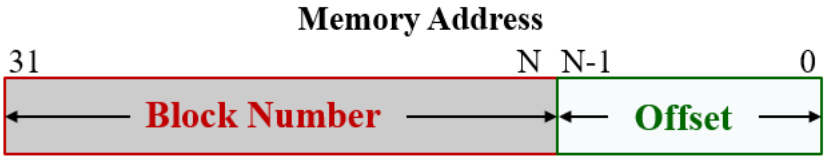
Cache Set Index
= (BlockNumber) modulo (NumberOfCacheSets)



Cache Block size = 2^N bytes
Number of cache sets = 2^M
Offset = N bits
Set Index = M bits
Tag = 32 – (N + M) bits

Observation:
It is essentially unchanged from the direct-mapping formula

10.8.3 Example

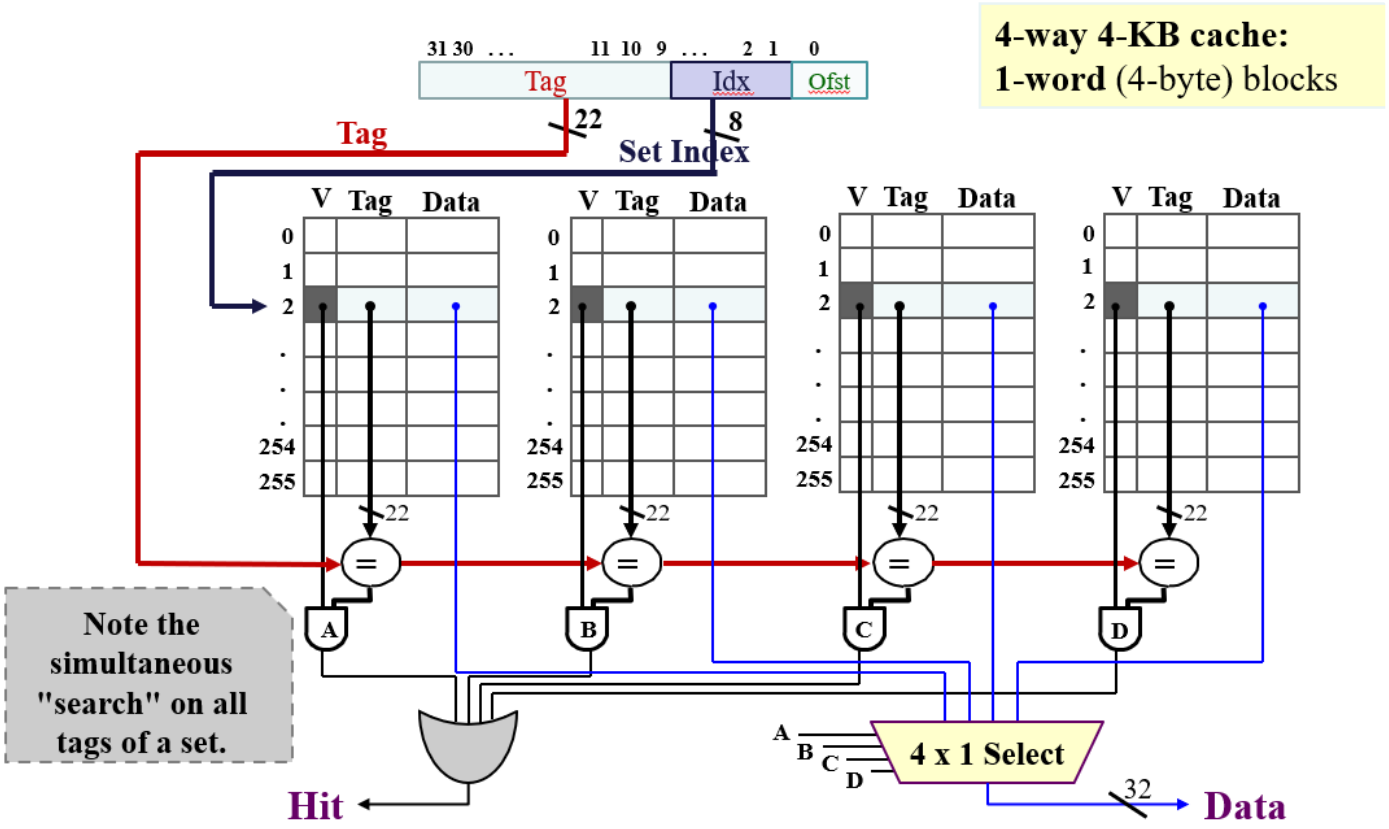


Offset, N = 2 bits
Block Number = 32 – 2 = **30 bits**
Check: Number of Blocks = 2^{30}

Number of Cache Blocks
= 4KB / 4bytes = 1024 = 2^{10}

4-way associative, number of sets
= 1024 / 4 = 256 = 2^8
Set Index, M = 8 bits

Cache Tag = 32 – 8 – 2 = **22 bits**



10.8.4 SA Cache Example

Setup

Given:

- Memory access sequence: `4,0,8,36,0`
- 2-way set-associative cache with a total of four 8-byte blocks (total of 2 sets)
- Indicate hit/miss for each access

Offset, N = 3 bits
Block Number = 32 – 3 = 29 bits

2-way associative, number of sets= 2 = 2^1
Set Index, M = 1 bits

Cache Tag = 32 – 3 – 1 = **28 bits**

Load #1

Load from `4`:

Check: Both blocks in Set 0 are invalid (cold miss)

Result: Load from memory and place in Set 0 - Block 0

> 原先index位表示的是写入哪一个block，现在由于在block上层还有set，所以这里的index是set index

Set Index	Block 0				Block 1			
	Valid	Tag	W0	W1	Valid	Tag	W0	W1
0	~~0~~ 1	0	M[0]	M[4]	0			
1	0				0			

Load #2

Load from `0`:

Result: Valid bit and Tags match in Set 0 - Block 0

Set Index	Block 0				Block 1			
	Valid	Tag	W0	W1	Valid	Tag	W0	W1
0	1	0	M[0]	M[4]	0			
1	0				0			

Load #3

Load from `8`:

Check: Both blocks in Set 1 are invalid

Result: Load from memory and place in Set 1 - Block 0

Set Index	Block 0				Block 1			
	Valid	Tag	W0	W1	Valid	Tag	W0	W1
0	1	0	M[0]	M[4]	0			
1	~~0~~ 1	0	M[8]	M[12]	0			

Load #4

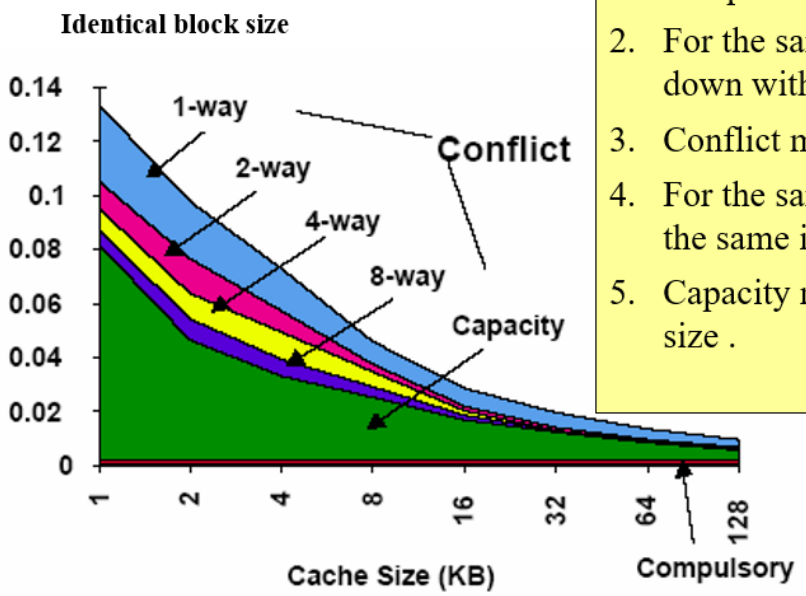
Load from `36`:

Check: Valid is 1 but tag mismatched in Set 0 - Block 0, while Set 0 - Block 1 is invalid

Result: Load from memory and place and place in Set 0 - Block 1

Set Index	Block 0				Block 1			
	Valid	Tag	W0	W1	Valid	Tag	W0	W1
0	1	0	M[0]	M[4]	~~0~~ 1	2	M[32]	M[36]
1	1	0	M[8]	M[12]	0			

10.9 Cache Performance



Observations:

1. Cold/compulsory miss remains the same irrespective of cache size/associativity.
2. For the same cache size, conflict miss goes down with increasing associativity.
3. Conflict miss is 0 for FA caches.
4. For the same cache size, capacity miss remains the same irrespective of associativity.
5. Capacity miss decreases with increasing cache size .

Total Miss = Cold miss + Conflict miss + Capacity miss
Capacity miss (FA) = Total miss (FA) – Cold miss (FA), when Conflict Miss➔0

10.10 Block Replacement Policy

Set Associative or Fully Associative Cache:

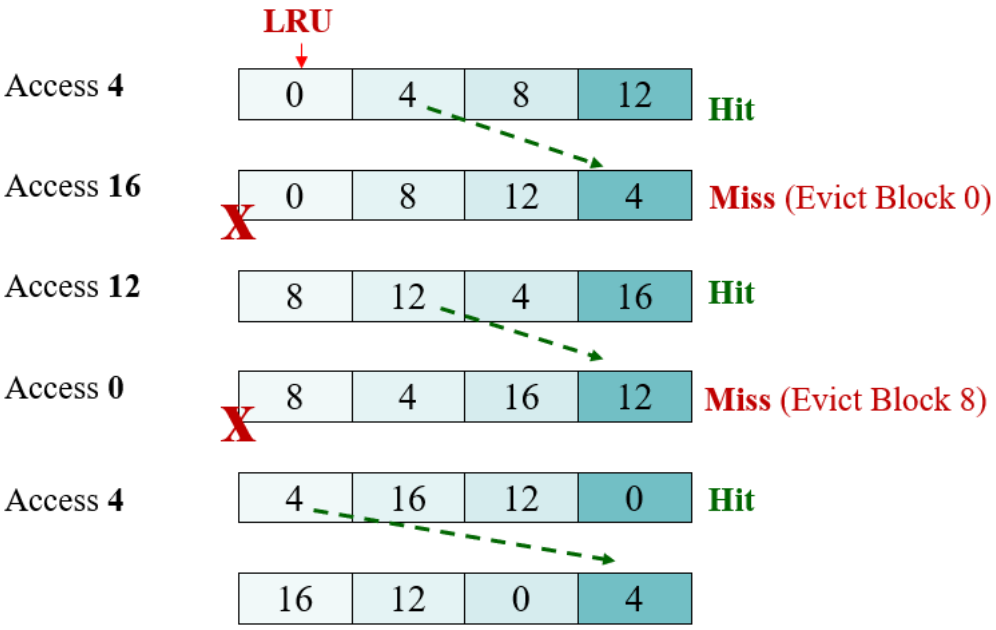
- Can choose where to place a memory block
- Potentially replacing another cache block if full
- Need block replacement policy

Least Recently Used (LRU)

- How: For cache hit, record the cache block that was accessed
  - When replacing a block, choose one which has not been accessed for the longest time
- Why: Temporal locality

LRU policy in action:

- 4-way SA cache
- Memory accesses: `0 4 8 12 4 16 12 0 4`



> Like a heap, the used block moved to the bottom. If a replacement is needed, replace the top block (least use)

Drawback for LRU:

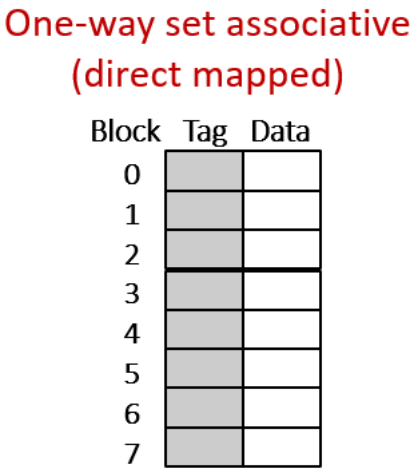
- Hard to keep track if there are many choices

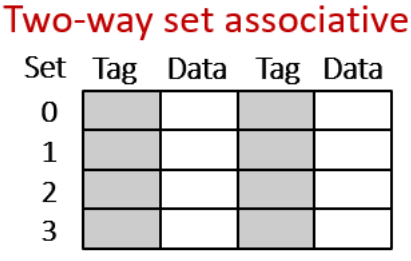
Other replacement policies:

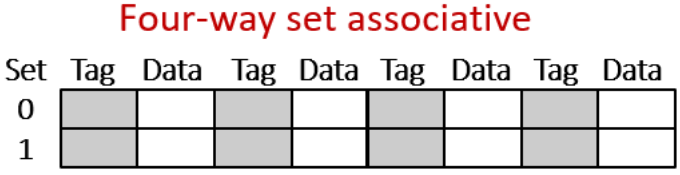
- FIFO
- Random replacement (RR)
- Least Frequently Used (LFU)

10.11 Summary

10.11.1 Cache Organizations







10.11.2 Cache Framework

Block Placement: Where can a block be placed in cache?

Direct Mapped:	N-way Set-Associative:
• Only one block defined by index	• Any one of the **N** blocks within the set defined by index

Block Identification: How is a block found if it is in the cache?

Direct Mapped:	N-way Set Associative:
• Tag match with only one block	• Tag match for all the blocks within the set

Block Replacement: Which block should be replaced on a cache miss?

Direct Mapped:	N-way Set-Associative:
• No Choice	• Based on replacement policy

Write Strategy: What happens on a write?

Write Policy: Write-through vs write-back

Write Miss Policy: Write allocate vs write no allocate

11 - Introduction to Operation Systems

11.1 Brief Introduction & Basic Terminology

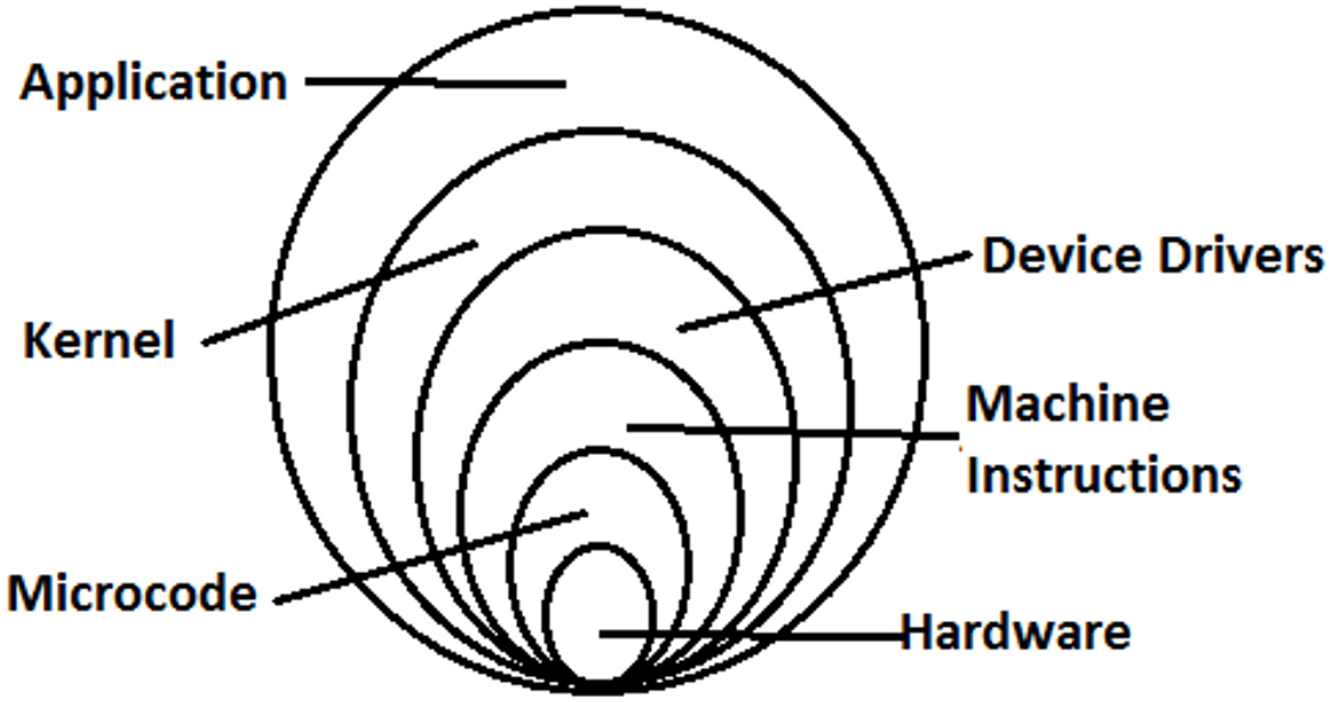
What are Operation Systems (OS)

- An “operation system” is a suite (i.e., a collection) of specialized software that:
  - Gives you access to the hardware devices like disk drives, printers, keyboards and monitors
  - Controls and allocate system resources link memory and processor time
  - Gives you the tools to customize your and tune your system
- Usually consists of several parts:
  - Bootloader - First program run by the system on start-up. Loads remainder of the OS kernel.
    - On Windows systems this is found in the Master Boot Record (MBR) on the hard risk
  - Kernel - The part of the OS that runs almost continuously
    - We will mainly foucus on this part
  - System Programs - Programs provided by the OS to allow:
    - Access to programs
    - Configuration of the OS
    - System maintenance, etc.

- **操作系统的定义**：操作系统是一套专业软件的集合，它负责以下任务：
  - **硬件设备的访问**：允许用户访问和控制硬件设备，如磁盘驱动器、打印机、键盘和显示器等。
  - **系统资源的控制与分配**：管理和分配系统资源，如内存和处理器时间。
  - **系统定制和调优工具**：提供工具，允许用户自定义和调整系统设置以适应其需求。
- **操作系统的组成部分**：操作系统通常包含几个主要部分：
  - **引导加载程序（Bootloader）**：是系统启动时运行的第一个程序。它负责加载操作系统的其余部分（即内核）。在Windows系统中，引导加载程序通常位于硬盘驱动器的主引导记录（MBR）中。
  - **内核（Kernel）**：是操作系统的核心部分，几乎持续不断地运行。内核负责管理系统级的操作，如进程管理、内存管理、设备驱动程序的执行等。
  - 系统程序（System Programs）

    这些是操作系统提供的程序，用于：
    - 访问程序。
    - 配置操作系统。
    - 进行系统维护等。

Basic Terminology



11.2 How an OS works

11.2.1 Bootstrapping

> Bootstrapping又被称为引导过程,是计算机启动时加载操作系统到内存中的过程.
>
> 当开启计算机时，CPU首先在一个预定的位置（如x86架构中的BIOS或UEFI固件）寻找启动指令。这个过程称为自检（POST），之后，控制权(系统执行指令的能力和权限)被交给引导加载程序（Bootloader）。在Windows系统中，这个引导加载程序通常位于硬盘的主引导记录（MBR）或者近年来使用的GUID分区表（GPT）的等效区域。
>
> 引导加载程序有责任：
>
> 1. 识别并初始化系统硬件。
> 2. 加载操作系统内核到内存中并执行它。
>
> 这个过程称为“引导”或“启动”，因为它好比是计算机通过其自身的引导带（bootstrap）来“提升”自己进入一个可操作的状态。这个术语来源于“自力更生”的表述，意指一个系统能够不依赖外部输入自主地启动。
>
> 在内核被加载并执行之后，它接管系统，初始化其它系统级别的软件，最终提供用户接口，如命令行或图形用户界面。这样，计算机便准备好接收用户输入，并执行程序了。

The OS is not present in memory when a system is “cold started”

- When a system is first started up, memory is completely empty
- We need to load system into memory

We start first with a bootloader

- Bootloader is a tiny program in the first (few) sector(s) of the hard-disk

    - The first sector is generally called the “boot sector” or “master boot record (MBR)” for this reason
- The bootloader’s job is to load up the main part of the OS and start it up

11.2.2 Process Management

Context Switching

> Context switching 是指操作系统内核在多个进程（或线程）之间切换执行权的过程。这是多任务操作系统进行任务管理的基本功能之一，允许单个处理器在多个任务之间迅速切换，从而给用户一种同时执行多个程序的错觉。
>
> 具体来说，在进行 context switching 时，操作系统会执行以下步骤：
>
> 1. **保存状态**：操作系统会保存当前正在执行的进程（或线程）的状态信息。这通常包括程序计数器、寄存器内容、系统调用状态、内存映射等。
> 2. **加载状态**：操作系统随后加载另一个进程（或线程）的状态信息到这些硬件组件中。这个过程包括更新程序计数器以指向新任务的代码位置，恢复寄存器的内容，以及设置内存访问权限等。
> 3. **执行新任务**：加载新状态后，处理器开始执行选中的新任务。

A shortage of cores

- A typical system today has two to four “cores”
    - Cores: CPU unit that can execute process
- While, we have much more than 4 process to run

To manage these processes, we share a core very quickly between multiple processes. (slicing)

Key points:

- Entire sharing must be transparent
- Processes can be suspended and resumed arbitrarily, which means it is not usually possible to build in this “sharing” into a process

Solution:

- Save the “context” of the process to be suspended
- Restore the “context” of the process to be re-started

Scheduling

We see that a single system may have multiple processing units (cores), but there will generally be many more processes than cores. We already settled this problem by `context switching`.

But how do we choose which process to run if several processes want to run?

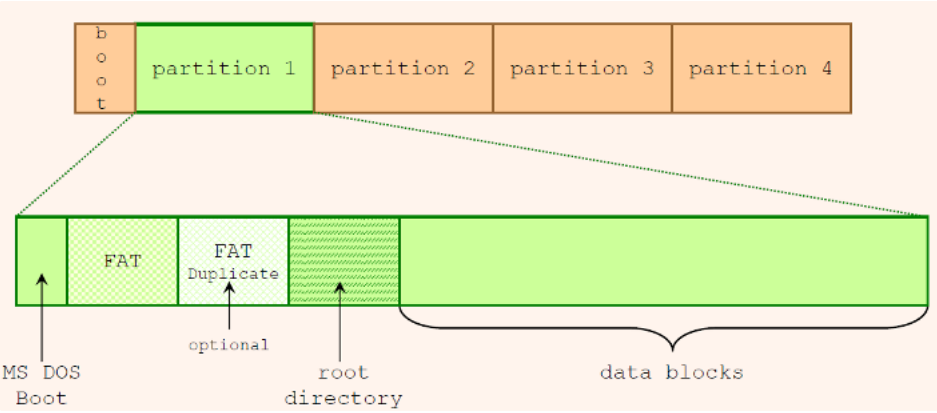
11.2.3 File Systems

An OS must support persistent storage

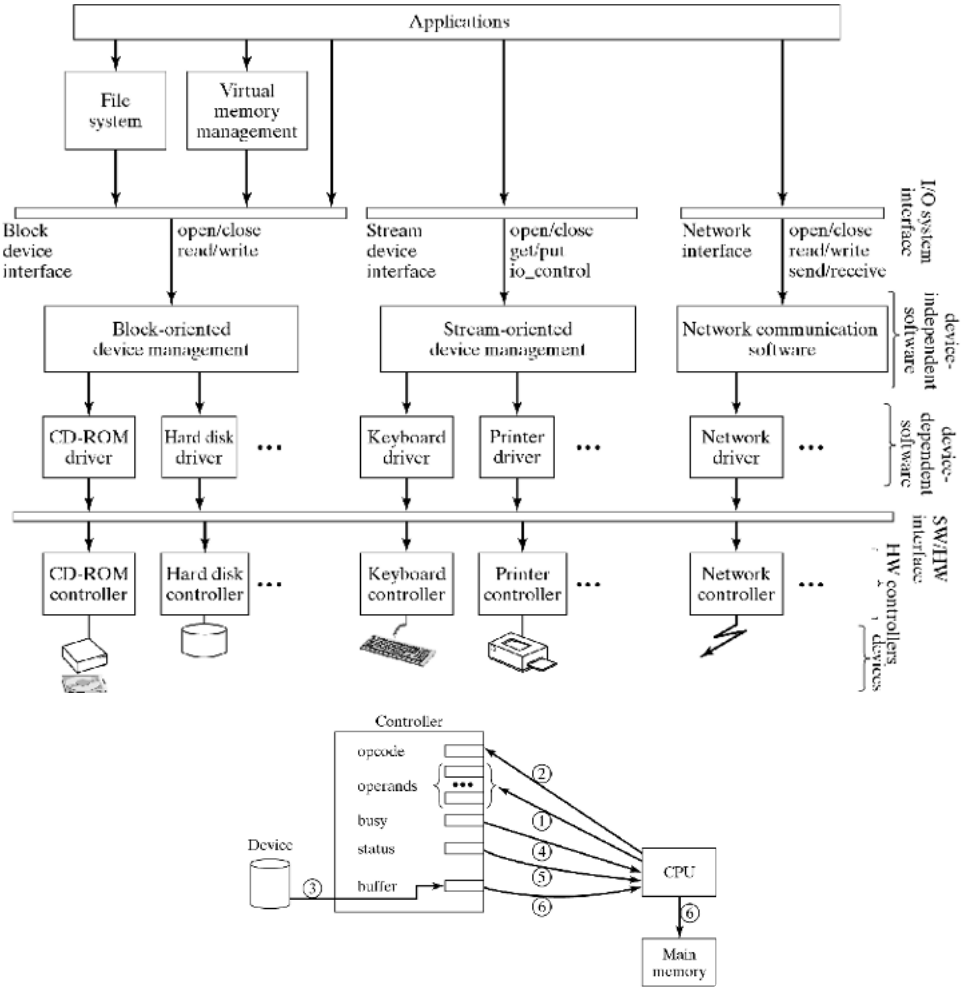
- This is storage whose contents do not disappear when the system is turned off

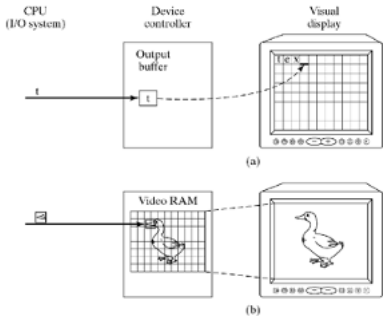
The primary way to do this is through a “file system” on persistent storage decices like hard drives.

- A set of data structures on disk and within the OS kernel memory to orgranize persistent data



11.2.4 Interfacing to Hardware





11.2.5 Memory Management

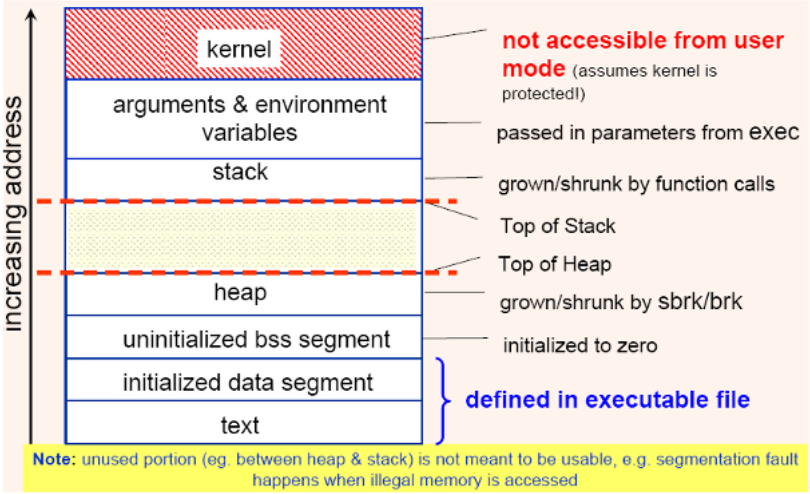
All programs require memory to work:

- To store instructions and (temp) data

OS must try to provide memory requested by the program

Note:

- Program can also ask for (and release) memory dynamically using `new`, `delete`, `malloc` (memory allocation function in C std library) and `free`



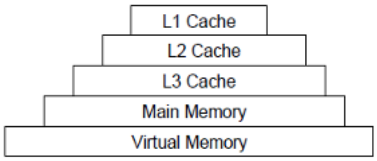
11.2.6 Virtual Memory Management

> Difference between memory and virtual memory:
>
> 1. **内存（物理内存或RAM）**：
>   - **直接访问**：物理内存是由实际的硬件内存条组成的。CPU可以直接对其进行访问。
>   - **有限的容量**：物理内存的容量是有限的，由安装在计算机上的内存条的大小决定。
>   - **快速存取**：物理内存的存取速度非常快，它是CPU执行程序和存储运行中程序数据的主要场所。
> 2. **虚拟内存**：
>   - **扩展内存容量**：虚拟内存是一种内存管理技术，它使得操作系统能够使用硬盘空间作为内存使用，从而扩展了可用的内存容量。
>   - **内存抽象**：虚拟内存通过分页（paging）或分段（segmentation）机制为每个程序提供了一个连续的内存地址空间，这是对物理内存的一种抽象。

    - **存取较慢**：虚拟内存使用硬盘来存储数据，因此它比物理内存慢得多。当系统物理内存不足时，操作系统会将部分数据从物理内存交换到虚拟内存中，这个过程称为换页（paging）或交换（swapping）。

物理内存是计算机中实际存在的硬件部分，而虚拟内存是一种软件层面的抽象，它使用硬盘空间来模拟更大的内存容量。虚拟内存允许计算机运行内存需求超过物理内存容量的程序，但这种超额部分的性能会因为硬盘的使用而降低。

For cost/speed reasons memory is organized in a hierarchy:



The lowest level is called “virtual memory”, and is the slowest but cheapest memory.

- Virtual memory using hard disk space to provide a contineous memory address space (using paging and segmentation)
- **Virtual memory is an abstraction for memory**
- Allow us to fit much more instructions and data than memory allows

11.2.7 Security

Here security means controlling access to various resources

- Data (files)
    - Encryption
    - Access control lists
- Resources
    - Access to the hardware (biometric, password)
    - Memory access
    - File access

11.3 Monolithic Kernels（整体内核）

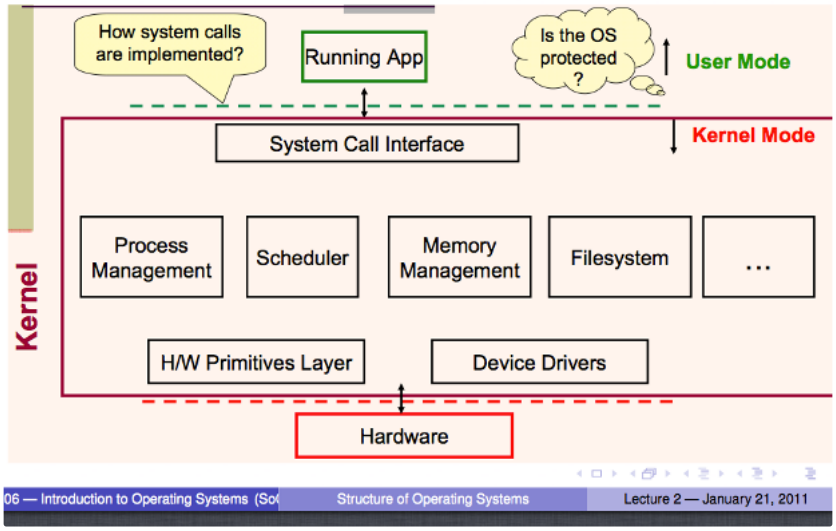
Monolithic Kernels（整体内核）是一种操作系统内核的设计方式，它将大部分的系统服务和驱动程序集成到内核空间中。这种设计与微内核（Microkernel）或层次内核（Layered Kernel）形成对比。

整体内核的特点包括：

- **集中式管理**：几乎所有的系统管理任务，如进程管理、内存管理、文件系统以及网络堆栈等，都在一个大的内核空间中执行。
- **性能**：因为服务和驱动程序在内核空间运行，它们可以直接访问硬件和内存，这通常会带来更好的性能，尤其是在系统调用和设备操作时。
- **简化的通信**：内核中的不同服务和模块之间可以直接进行函数调用，而无需复杂的消息传递机制。
- **安全与稳定性风险**：如果内核的任何部分发生故障，整个系统可能会受到影响。这意味着整体内核可能比微内核更容易受到单点故障的影响。
- **可移植性和维护性**：整体内核因为其复杂性和庞大的代码基础，可能在可移植性和维护性方面面临挑战。

- Kernals can be monolithic or microkernel
- Monolithic kernels:
  - All major parts of the OS - devices drivers, file systems, IPC, running in “kernel space”
    - Kernel space generally means an elevated execution mode where certain privileged operations are allowed
  - Bits and pieces of the kernel can be loaded and unloaded at runtime (using `modprobe` in Linux)
  - Examples of monolithic kernels: Linux, Windows

> 整体内核的主要特征：
>
> - **操作系统的主要部分在内核空间运行**：这包括设备驱动程序、文件系统、进程间通信（IPC）等。这些都是操作系统的核心组件，它们在内核空间运行，这是一个有高权限的执行环境。
> - **内核空间**：内核空间是指保留给操作系统内核的内存区域，并且在这个空间中运行的代码可以执行特权操作，如直接访问硬件或管理系统资源。在大多数现代操作系统中，内核空间与用户空间相隔离，后者是应用程序运行的环境。
> - **内核的模块化**：尽管是整体内核，但现代操作系统（如Linux）允许某些内核组件以模块化的形式存在，可以在运行时动态加载和卸载。在Linux中， `modprobe` 工具用于管理这些内核模块：加载新的模块以增加功能，或者卸载模块以释放资源或因为不再需要某个特定的硬件支持。
>
> 通过动态加载和卸载内核模块，系统管理员可以根据需要调整系统的硬件支持和功能，而无需重启系统。这提供了一种灵活性，使得整体内核的系统能够更加适应不断变化的硬件和软件环境。



11.4 Microkernels

In modular kernels:

- Only the “main” part of the kernel is in “kernel space”
  - Which contains the important stuff like the scheduler, process management and memory management
- The other parts of the kernel operate in “user space” as system services
  - The file systems, device drivers

Example of microkernels: MacOS

> 在微内核设计中，只有核心的功能部分运行在内核空间，而其他部分则可以以系统服务的形式运行在用户空间。
>
> 具体来说：
>
> - **核心内核在内核空间**：微内核中，只有最关键的组成部分，如调度器（负责决定哪个进程获得CPU时间）、进程管理（负责创建和终止进程）、内存管理（负责分配和回收内存资源）等运行在内核空间。这部分内核是始终加载的，因为它们对于操作系统的运行至关重要。
> - **其他内核部分在用户空间**：与整体内核不同，微内核中的其他组件，如文件系统和设备驱动程序，可能作为系统服务在用户空间中运行。这意味着它们虽然是内核功能的一部分，但它们的执行环境与普通的用户级应用程序相同。
>
> 这种设计的优势在于：
>
> - **安全性和稳定性**：由于非核心组件在用户空间运行，它们即使失败也不太可能导致整个系统崩溃。这增加了系统的整体稳定性。
> - **灵活性**：在用户空间运行的系统服务可以像普通应用程序一样启动和停止，不需要重新启动内核来更改这些服务。

11.5 External View of an OS

The kernel itself is not very useful

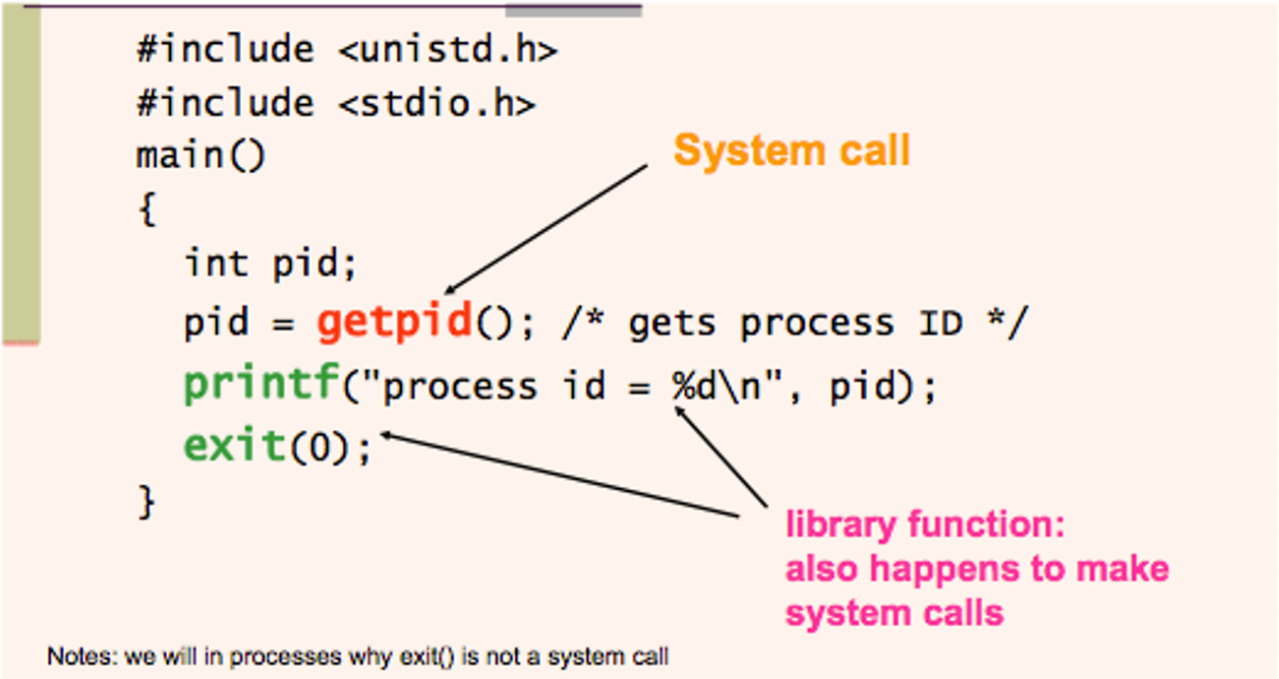
- Provides key functionality, but need a way to access all this function
  We need other components:
- System libraries (`stdio` , `unistd` , etc)
- System services (creat, read, write, ioctl, sbrk, etc)
- OS Configuration (task manager, setup, etc)
- System programs, (Xcode, vim, etc)
- Shells
- Admin tools
- User applications

11.6 System Calls

System calls are calls made to the "Application Program Interface" (API) of the OS.

- UNIX and similar OS mostly follow the POSIX standard
  - Based on C
  - Programs become more portable

- Windows follows the WinAPI standard



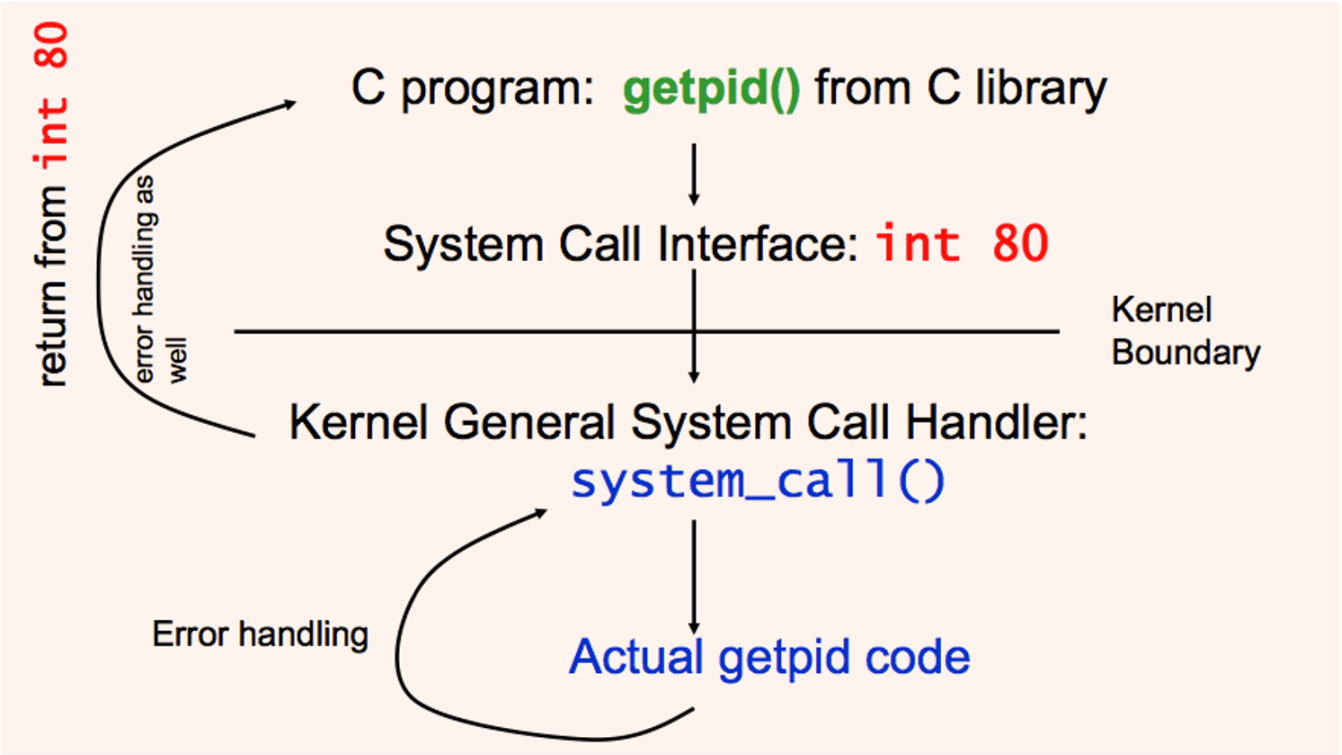
11.7 User Mode + Kernel Mode

- Want to protection between kernel and executing program
- Program (actually process) runs in user mode
- During system call - running kernel code in kernel mode
- After system call, back to user mode

How to switch mode?
Use privilege mode switching instructions:

- Syscall instruction

- Software interrupt - instruction which raises specific interrupt from software



- **user mode**: (outside kernel)
  - C function wrapper (e.g. `getpid()`) for every system call in C library (not really the real system call, sometimes loosely call it a system call but **not** technically correct)
  - assembler code to setup system call no, arguments
  - trap to kernel (the real system call after all arguments setup)
- **kernel mode:** (inside kernel)
  - dispatch to correct routine
  - check arguments for errors (eg. invalid argument, invalid address, security violation)
  - do requested service
  - return from kernel trap to user mode
- **user mode**: (outside kernel)
  - returns to C wrapper – check for error return values

12 - Process Management

12.1 Program vs. Process

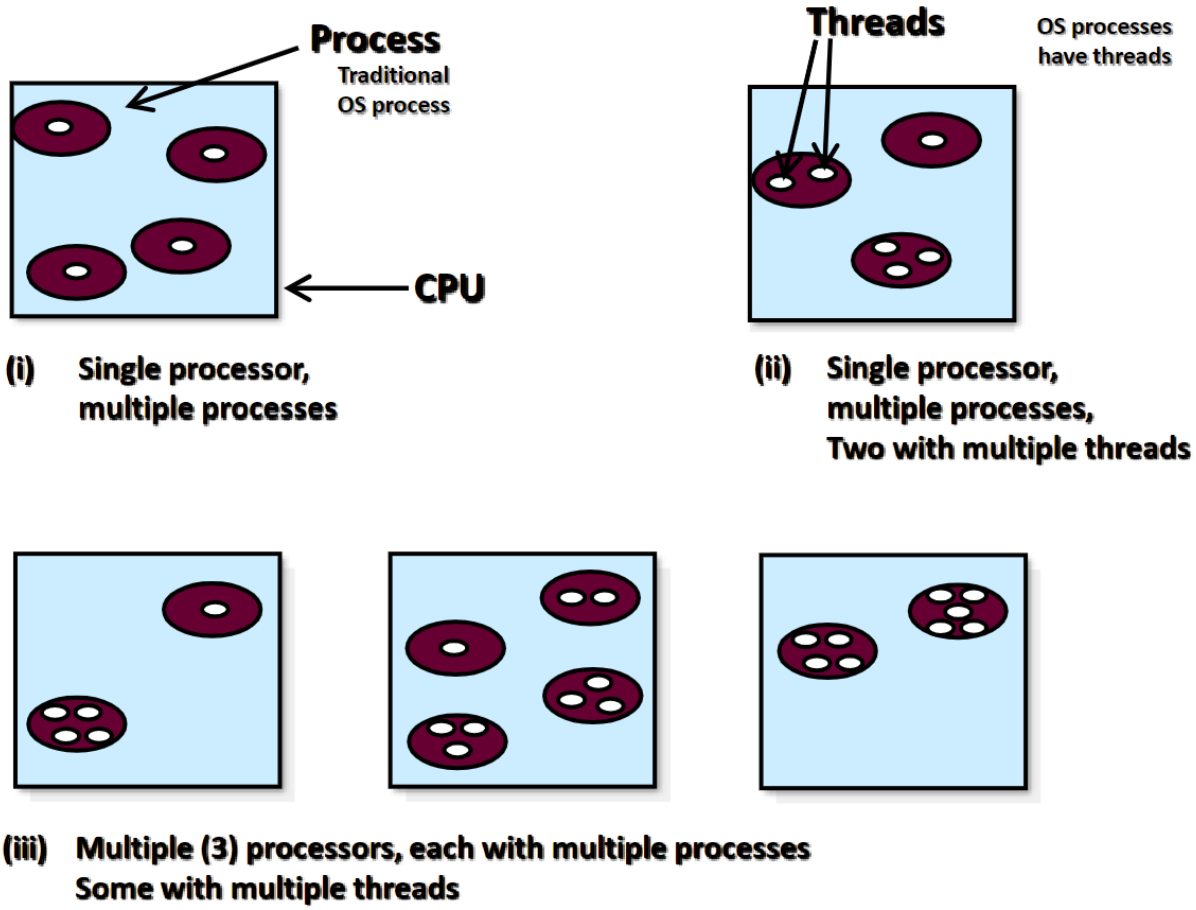
A program consists of:

- Machine instructions (and possibly source code)
- Data
- A program exist as a file on the disk. E.g. `command.exe`
  A process consists of:
- Machine instructions (and possibly source code)
- Data
- Context
- Exists as instructions and data in **memory**
- MAY be executing on the CPU

> 一个程序（program）是一组指令，它被写入计算机系统的存储器中，以便在需要时执行。程序是一种静态的实体，它包含了处理任务所需的所有指令和数据，但它本身并没有具体的执行状态。
> 然而，进程（process）是执行中的程序的实例。当一个程序被加载到计算机的内存并开始执行时，它成为一个进程。进程是动态的，它有自己的执行状态和运行的上下文。每个进程都有自己的内存空间、程序计数器（记录下一个将被执行的指令的地址）和一组寄存器等。进程可以独立地执行，并与其他进程并行运行。每个进程都有独立的资源分配和管理。
> 总结起来，程序是一组指令和数据的集合，而进程是程序在执行过程中的实例，具有独立的执行状态和资源。

- A single program can produce multiple processes
  - E.g., `chrome.exe` is a single program

- But every tab in Chrome is a new process

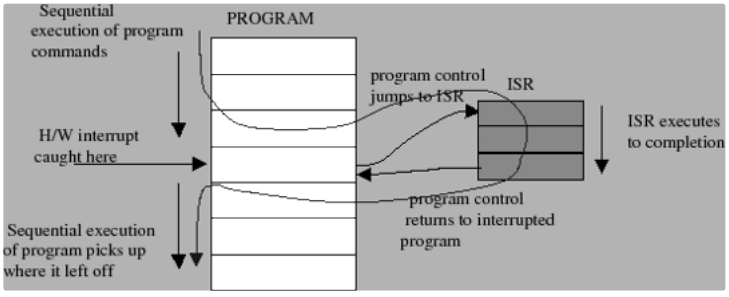


12.2 Interrupts

> Definition:
> IRQ信号：是Interrupt ReQuest(中断请求)的简称.他是硬件设备用来通知CPU其需要处理某些事件的一种机制。每个IRQ信号都对应于计算机中的特定硬件设备或内部系统。当设备需要CPU的注意时，其会发送一个IRQ信号给CPU。

When a device needs attention from the CPU, it triggers what is called an "interrupt":

- Each device is connected to the CPU via an input called an "Interrupt Request" or IRQ line
  - When a device needs attention, it pulls the IRQ line HIGH or LOW (dependin on whether line is "active high" or "active low")
- This line is checked at the end of WB (Write Back) stage in the CPU Execution Cycle
- Diagrams below show the CPU's program execution flow when an interrupt occurs



- If line has been pulled, CPU interrupts the code it is currently running to run code to attend to the device
  - Current PC (Program Counter) is pushed onto the stack
  - CPU consults and "interrupt vector table" to look for the address of the "interrupt service routine" or ISR - a small bit of code that will read/write/tend to the device
  - This address is loaded into PC, and the CPU starts executing the handler
  - When the handler exits, the previous PC value is popped off the stack and back into PC, and execution resumes at the interrupted point

中断是硬件设备通知CPU它需要处理某个事件的一种机制。

- **中断请求（IRQ）**：每个设备通过一个称为中断请求（Interrupt Request，简称IRQ）的输入线与CPU连接。当设备需要CPU的注意时，它会将IRQ线拉高或拉低，这取决于线路是活动高（active high）还是活动低（active low）。
- **CPU执行周期的WB阶段**：在CPU的执行周期中，写回（Write Back，简称WB）阶段是最后一个阶段，在这个阶段结束时，CPU会检查IRQ线。
- **中断的处理**：如果检测到IRQ线被激活，CPU会中断当前正在执行的代码，转而执行处理该设备的代码。这个过程包括以下步骤：
- **保存程序计数器（PC）**：当前的程序计数器（PC），它指示CPU当前执行到程序的哪个位置，会被推入（push）栈中以便之后能回到中断前的执行点。
- **查询中断向量表**：CPU会查看一个特殊的表——中断向量表，以找到对应的中断服务例程（Interrupt Service Routine，简称ISR）的地址。ISR是一段专门用来处理特定设备请求的小代码。
- **执行中断处理程序**：找到ISR的地址后，这个地址被加载到PC中，然后CPU开始执行这个中断处理程序
- **恢复执行**：中断处理程序执行完毕后，之前保存的PC值会从栈中弹出（pop），重新加载到PC中，CPU随后会从被中断的地方继续之前的程序执行。

这个机制允许CPU在不同的任务和设备请求之间高效切换，确保对紧急事件的快速响应，同时也保持程序执行的连贯性。

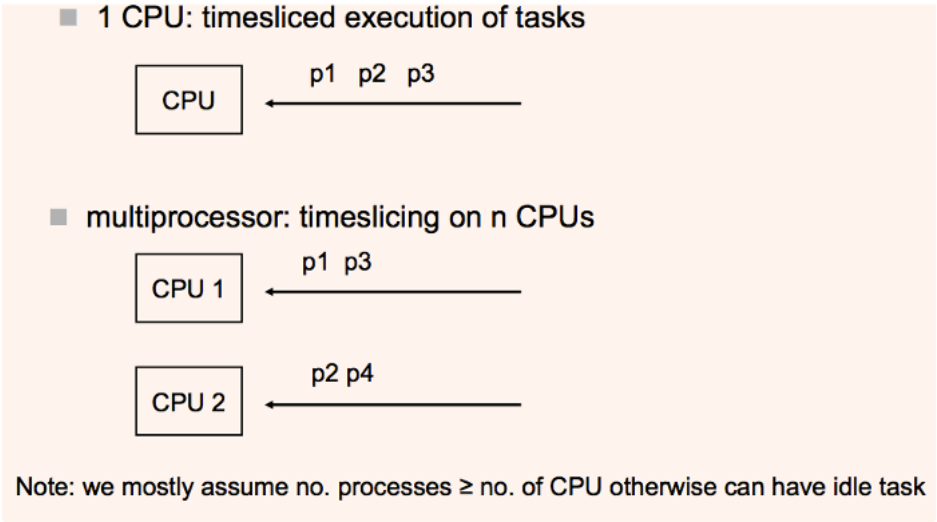
- The CPU asserts the interrupt acknowledge (IA) line to tell the device that its request has been handled
  - Sometimes the CPU will de-assert the IRQ line instead of employing a separate IA line
- Interrupts are key to allowing us to run multiple processes on a CPU:
  - A hardware device called a “timer” will interrupt the CPU every ms
  - Interrupt handler will switch to a new process

- **中断确认（IA）信号**：当CPU开始处理一个设备的中断请求时，它会发出一个中断确认（Interrupt Acknowledge，简称IA）信号告知该设备它的请求已经被接受。这样设备就知道它的信号已经被CPU注意到，并将开始被处理。
- **取消IRQ信号**：有时候，CPU处理完中断请求后，会直接取消（de-assert）IRQ线，而不是使用单独的IA线。这个行为是为了清除中断请求，让系统知道该请求已经得到处理，以便设备知道不需要继续发出中断请求。
- **中断在多进程运行中的作用：**

- **定时器**：硬件设备如定时器（timer）会周期性地（例如，每毫秒）中断CPU。这种中断被用于操作系统的时间管理和进程调度。
- **中断处理程序切换进程**：当定时器中断发生时，中断处理程序会执行，它可能会选择停止当前运行的进程并切换到另一个进程。这是实现时间共享和多任务处理的关键机制

12.3 Execution Modes

- Programs usually run sequentially
  - Each instruction is executed one after other
- Having multiple cores or CPUs allow parallel (concurrent) execution
  - Streams of instructions with no dependencies are allowed to executed together
- A multitasking OS allows several programs to run concurrently
  - Interleaving, or "time slicing"

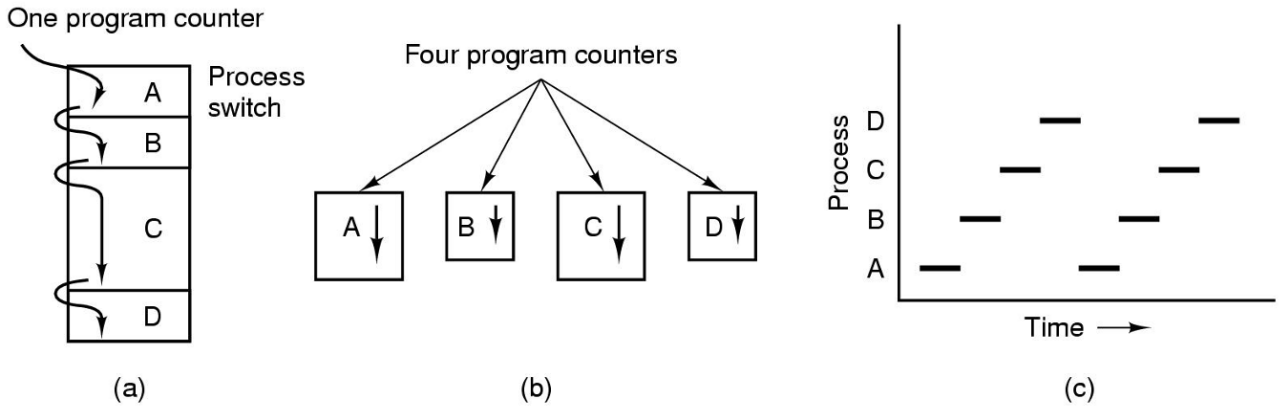


12.4 Processes and Process Management

12.4.1 The Process Model

- We will assume a single processor with a single core.
  - This is a legitimate assumption because in general the number of processes >> the number of cores
  - So each core must still switch between processes
- In single-core single processor:

- At any one time, at most one process can execute



- Figure (b) shows what "appears" to be happening in a single processor system running multiple processes:
  - There are 4 processes each with its own program counter (PC) and registers
  - All 4 processes run independently of each other at the same time
- Figure (a) shows what actually happens
  - There is only a single PC and a single set of registers
  - When one process ends, there is a "context switch" or "process switch":
    - PC, all registers and other process data for Process A is copied to memory
    - PC, register and process data for Process B is loaded and B starts executing
  - Figure (c) illustrates how processes A to D share CPU time

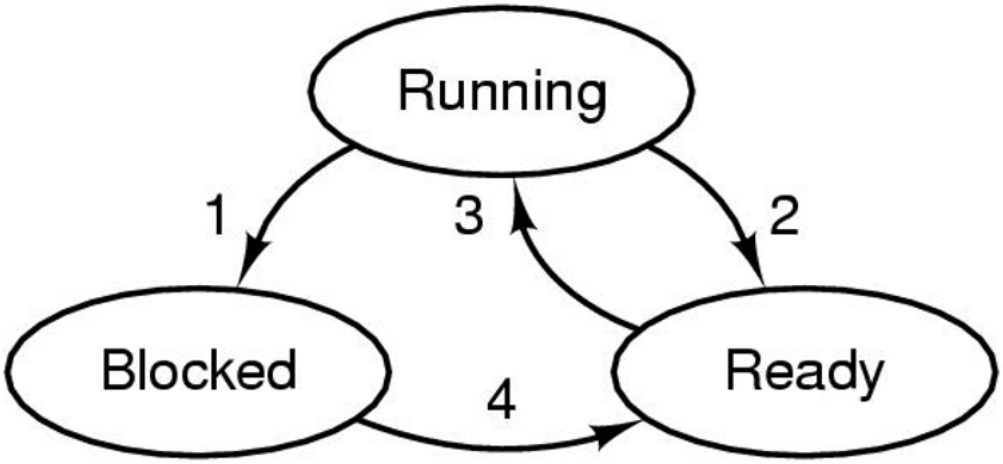
图b是虚假的单核处理器中多个进程的运行模式。图a和图c是真实的。在实际情况下，单核处理器中运行多个进程时，多个进程共享程序计数器和寄存器。

图a和b的运行模式是不一样的。在a中，其运行模式为运行完一个进程后，再运行下一个进程。在图c中，其使用了slicing，即将多个进程分为了几个小的切片，在运行时间中多次，重复的运行每个进程的切片。由于每个运行时间循环极短，所以在用户视角中，这和并行运行没有区别。

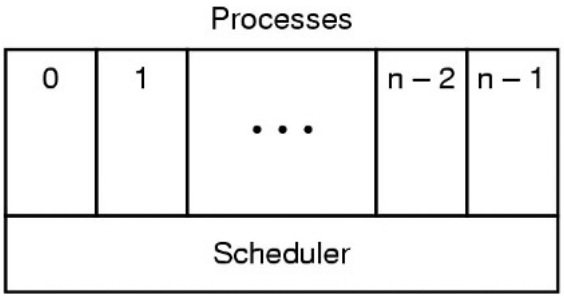
12.4.2 Process States

- A process can be in one of 3 possible states:
  - Running
    - The process is actually being executed on the CPU
  - Ready
    - The process is ready to run but not currently running
    - A "scheduling algorithm" is used to pick the next process for running
  - Blocked
    - The process is waiting for "something" to happen so it is not ready to run yet
    - E.g., include waiting for inputs from another process
- The diagram below shows the 3 possible states and the transitions between them
  1. Process blocks for input
  2. Scheduler picks another process

3. Input becomes available



- The figure below shows how the processes are organised
  - The lowest layer selects (schedules) which process to run next
    - This is subject to "scheduling policies"



12.4.3 Switching between Processes

- When a process runs, the CPU needs to maintain a lot of information about it. This is called the "process context"
  - CPU register values
  - Stack pointers
  - CPU Status Word Register
    - This maintains information about whether the previous instruction resulted in an overflow or a "zero", whether interrupts are enabled
    - This is needed for branch instructions - assembly equivalents of `if` statements

The AVR Status Register – SREG – is defined as:

Bit	7	6	5	4	3	2	1	0	
0x3F (0x5F)	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- All of these values change as a process runs
- When a process is blocked or put into a READY state, a new process will be picked to take control of the CPU
  - All the information for the current process must be saved
  - The information for the new process must be loaded into the registers, stack pointer and status registers

    - This is to allow the new process to run like as though it was never interrupted
- This process is known as "context switching"

进程上下文包括：

- **CPU寄存器值**：这些是当前任务的中间计算结果、程序计数器、指令寄存器等。
- **栈指针**：指向进程栈顶部的指针，进程栈存储了执行路径、局部变量等。
- **CPU状态字寄存器**：包含标志位，指示上一个操作是否产生了溢出、是否结果为零、以及中断是否被允许等状态信息。

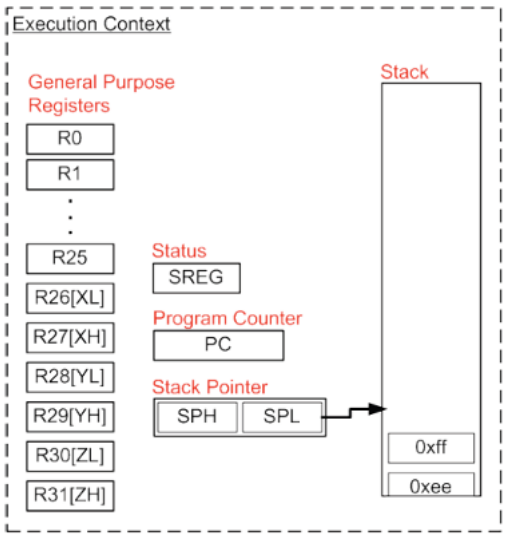
当进程运行时，所有这些值都会随着进程的执行而改变。如果一个进程被阻塞或者被置于就绪（Ready）状态，操作系统会选择另一个进程来接管cpu。此时，必须完成以下步骤：

1. **保存当前进程的上下文**：操作系统会保存当前进程的所有寄存器值、栈指针和CPU状态字寄存器等信息。这样做是为了保证当前进程在未来某个时刻能够恢复执行，就如同它从未被中断过一样
2. **加载新进程的上下文**：然后，操作系统会将下一个要执行的进程的上下文信息加载到CPU的寄存器、栈指针和状态寄存器中。这允许新进程从它上次停止的地方继续执行

这个从一个进程的上下文切换到另一个进程的上下文的过程称为“上下文切换”。上下文切换是操作系统用来分享处理器时间，实现并发执行多个进程的机制。尽管上下文切换是非常快速的，但它涉及到一些开销，因为保存和加载进程状态需要时间，因此操作系统设计时会试图最小化不必要的上下文切换以提高效率。

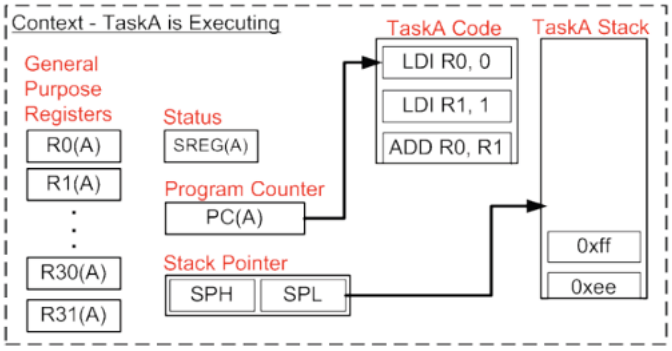
12.5 Context Switching on the FreeRTOS Atmega Port

- Each process is allocated a stack
    - Stack = Process
- The diagram shows the complete Atmega context
    - Registers R0-R31, PC
    - Status register SREG
    - Stack pointer SPH/SPL



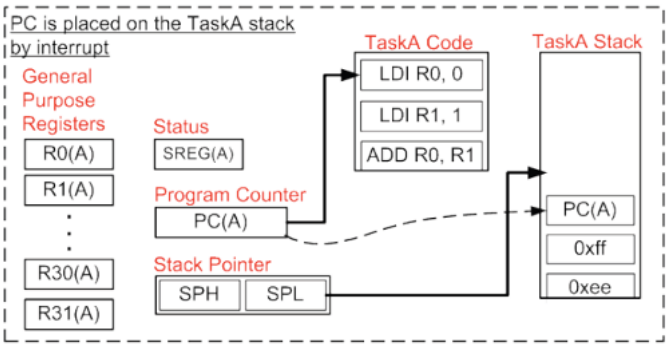
1. Task A interrupts

- Assume that at first Task A is executing
  - PC would be pointing at Task A code, SPH/SPL (stack pointer) pointing at Task A stack, Registers R0-R31 contain Task A data



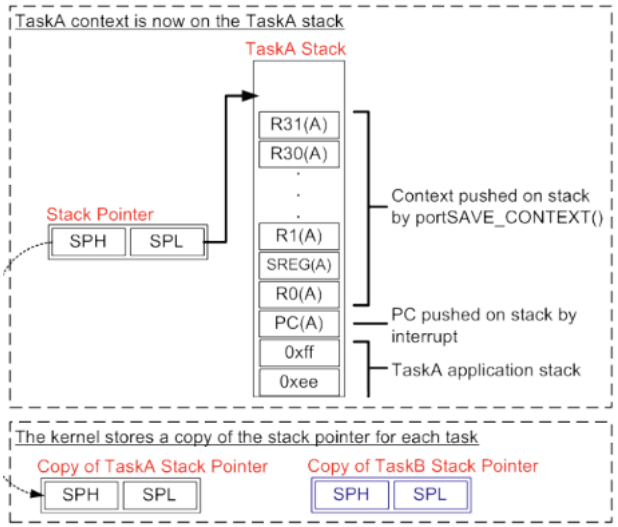
Push Program Counter (PC) into the Task A stack

- FreeRTOS relies on regular interrupts from Timer 0 every ms to switch between tasks. When the interrupt triggers, PC is placed onto Task A's stack.



Push Program Context into the stack

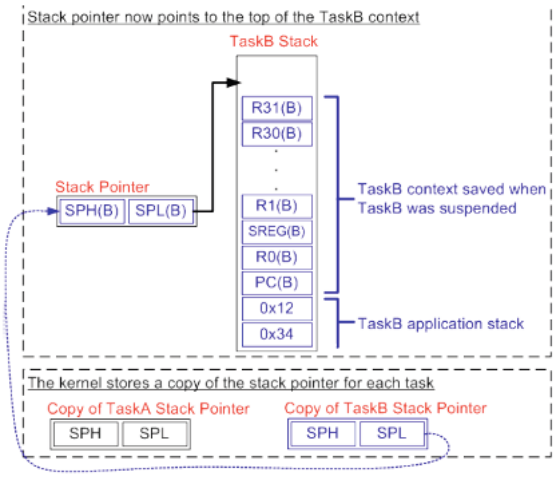
- The ISR calls `portSAVECONTEXT`, resulting in Task A's context being pushed onto the stack
- `pxCurrentTCB` will also hold SPH/SPL after the context save
  - This must be saved by the kernel



2. Run Task B

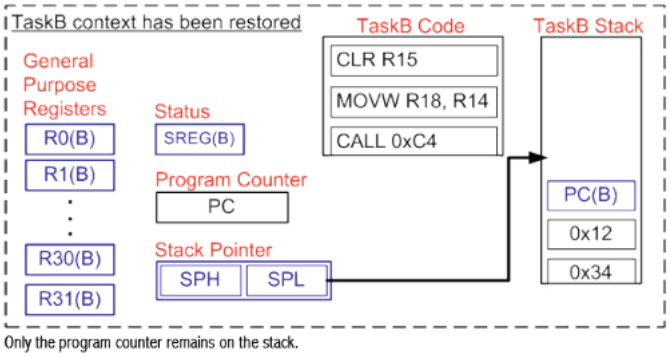
Point Stack Pointer (SP) to Task B's Stack

- The kernel then selects Task B to run, and copies its SPH/SPL values into `pxCurrentTCB` and calls `portRESTORE_CONTEXT` to restore the process context of Task B
  - The first two lines will copy `pxCurrentTCB` into SPH/SPL, causing Stack Pointer to point to Task B's stack



Pop Process Context From Stack

- The reset of `portRESTORE_CONTEXT` is executed, causing Task B's data to be loaded into registers R0-R31 and SREG
  - Now Task B can resume like as though nothing happended



- The reverse operation is `portRESTORE_CONTEXT`. The stack pointer for the process being restored must be in `pxCurrentTCB`
- Only Task B's PC remains on the stack. Now the ISR exists, causing this value to be popped off onto the AVR's PC
  - PC points to the next instruction to be executed
  - End result: Task B resumes execution, with all its data and SREG correct

12.5 Process Creation

- A process can be created in Python by using a `fork()` call:

```python
# Python code to create child process
import os

def parent_child():
    n = os.fork()

    # n greater than 0  means parent process
    if n > 0:
        print("Parent process and id is : ", os.getpid())

    # n equals to 0 means child process
    else:
        print("Child process and id is : ", os.getpid())

# Driver code
parent_child()
```

- The creating process is called a "parent" process, while the created process is called "child" process
- When you run a program in your OS shell, the shell uses the OS to create a new process, then run the program in the new process
- In Python this is done by using `subprocess.call()`

```python
import subprocess

subprocess.call(["ls", "-lha"])
```

- The launched is thus a child process of the shell
- Ultimately, all UNIX process are children of "init", the main starting process in UNIX

12.6 Process Control Blocks

- When a process is created, the OS also creates a data structure to maintain information about that process:
  - Called a "Process Control Block" (PCB) and contains:
    - Process ID (PID)
    - Stack Pointer
    - Open Files
    - Pending Signals
    - CPU usage
  - PCB is stored in a table called a "Process Table"
    - One Process Table for entire system
    - One PCB per process
- When a process terminates:
  - Most resources like open files, etc., can be released and returned to the system
  - However the PCB is retained in memory:
    - Allows child processes to return results to the parent
  - Parent retrieves the results using a "wait" function call, afterwhich the PCB is released
- What if the parent never calls "wait"?
  - PCB remains in memory

- Child becomes a "zombie" process. Eventually process table will run out of space and no new process can be created
  当一个进程被创建时，操作系统会为其创建一个称为“进程控制块”（Process Control Block，简称PCB）的数据结构，用以维护该进程的相关信息。以下是PCB包含的关键信息：

1. **进程标识符（Process ID，PID）**：
   - 每个进程有一个唯一的标识符，用于区分系统中的不同进程。
2. **栈指针（Stack Pointer）**：
   - 指向进程的栈顶，栈用于存储函数参数、返回地址以及局部变量。
3. **打开文件（Open Files）**：
   - 进程打开的文件列表，通常包括文件描述符和相关的文件状态信息。
4. **挂起信号（Pending Signals）**：
   - 待处理的信号集，信号是进程间通信的一种方式。
5. **CPU使用情况（CPU usage）**：
   - 该进程使用CPU的统计信息，如累计CPU时间。
     PCB存储在一个称为“进程表”（Process Table）的结构中，系统中的每个进程都有一个对应的PCB。
     当进程终止时：

- 操作系统通常会释放大多数资源，如关闭打开的文件等，并将这些资源返回给系统。
- 然而，PCB会保留在内存中，这允许子进程将结果返回给父进程。
- 父进程通过调用“wait”函数来检索子进程的结果。在这之后，子进程的PCB被释放。
  如果父进程从未调用“wait”：
- 子进程的PCB会保留在内存中，这样的子进程被称为“僵尸进程”（Zombie Process）。
- 如果僵尸进程过多，进程表可能会耗尽空间，导致无法创建新的进程。

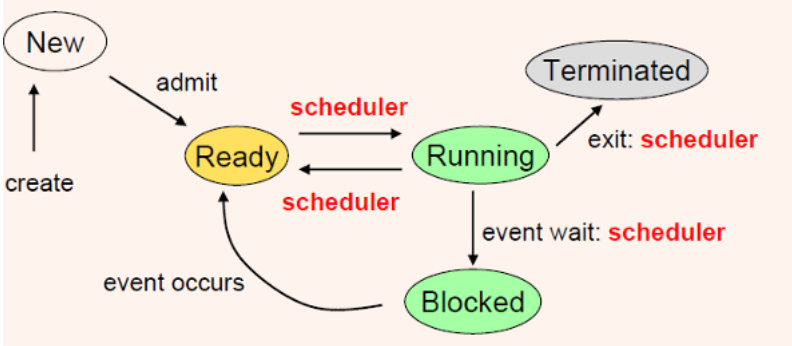
13 - Process Scheduling

13.1 Scheduling Environment

- Processes can be:
  - CPU bound
    - Most of the time spent on processing on CPU
    - Graphics-intensive applications are considered to be "CPU bound"
    - Multi-tasking opportunities come from having to wait for processing result
  - I/O bound
    - Most of the time spent on communicating with I/O devices
    - Multitasking opportunities come from having to wait for data from I/O devices

13.2 Process States

- Processes switch between a fixed set of states depending on events that take place
  - Scheduler is invoked at various points as shown below



13.3 Generic Scheduler Program

```
schedule() {
    while (queue not empty) {
        task = pick task from ready queue; //policy dependent
        delete task from queue
        switch to task
    }
}
```

> How to determine policies to pick the next task?

13.4 Type of Multitaskers

- Policies are determined by the kind of multitasking environment
  - **Batch Processing**
    - Not actually multitasking since only one process runs at a time to completion
  - **Co-operative Multitasking**
    - Currently running processes cannot be suspended by the scheduler
    - Processes must volunteer to give up CPU time
  - **Pre-emptive Multitasking**
    - Currently running processes can be force suspended by the scheduler
  - **Real-Time Multitasking**
    - Processes have fixed *deadlines* that must be met
      - If don't meet the deadline:
        - *Hard Real Time Systems*: System fails
        - *Soft Real Time Systems*: Mostly just an inconvenience. Performance of system degraded.

批处理（Batch Processing）：任务逐一执行，直至完成。在一个任务完成之前不会启动另一个任务
协作多任务处理（Co-operative Multitasking）：在这种环境中，当前正在运行的进程不会被操作系统的调度器强制挂起。进程必须自愿放弃CPU时间，使得其他进程有机会运行。
抢占式多任务处理（Pre-emptive Multitasking）：在这种模式下，操作系统的调度器可以强制挂起当前运行的进程，以便其他进程可以使用CPU。这样可以确保所有进程都能获得执行的机会。
实时多任务处理（Real-Time Multitasking）：这种模式要求进程必须在固定的截止时间前完成。如果进程未能在截止时间前完成：
- **硬实时系统**（Hard Real-Time Systems）：系统死机
- **软实时系统**（Soft Real-Time Systems）：通常只是造成不便，系统性能会降低

Scheduling Policies for Multitaskers

- Scheduling policies enforce a priority ordering over processes
  - As mentioned before, determined by multitasking type
- Example policies
  - Simplest policy (Great for *all type* of multitaskers)
    - **Fixed Priority**
  - Policies for *Batch Processing*
    - **First-come First-served** (FCFS)
    - **Shortest Job First** (SJF)
  - Policies for Co-operative Multitaskers
    - **Round Robin with Voluntary Scheduling** (VC)
  - Policies for Pre-emptive Multitaskers
    - **Round Robin with Timer** (RR)

        - **Shortest Remaining Time** (SRT)
    - Policies for Real-Time Multitaskers
        - **Rate Monotonic Scheduling** (RMS)
        - **Earliest Deadline First Scheduling** (EDF)

> **固定优先级** (Fixed Priority): 根据设置的优先级顺序来运行任务。
> **先来先服务** (First-come First-served, FCFS) : 按照任务到达的顺序来运行任务。
> **最短作业优先** (Shortest Job First, SJF) : 优先执行预计运行时间最短的任务
> **循环轮询与自愿调度** (Round Robin with Voluntary Scheduling, VC) : 任务轮流获得CPU时间，但每个任务在使用CPU时必须自愿放弃CPU时间，从而允许下一个任务运行
> **带时间切片的循环轮询** (Round Robin with Timer, RR): 为每个进程分配时间切片，在时间切片结束时调度器将其挂起，并将CPU分配给下一个进程
> **最短剩余时间** (Shortest Remaining Time, SRT): CPU被分配给(预估)剩余时间最短的进程，这是对SJF的改进

Fixed Priority Policy

- This is a simple policy that can be used across any type of multitasker
    - Each task is assigned a priority by the programmer
        - Usually priority number 0 has the highest priority
    - Tasks are queued according to priority number
    - Bath, Co-operative
        - Task with highest priority is picked to be run next
    - Pre-emptive, Real-Time
        - When a higher priority task becomes ready, current task is immediately suspended and higher priority task is run

Batch Scheduling Policies

Here we have two policies, FCFS and SJF

First Come First Serve, FCFS

- Arriving jobs are stored in a queue
- Jobs are removed in turn and run
- Particularly suited for bath systems
- Extension for interactive systems:
    - Jobs removed for running are put back into the back of the queue
    - This is also known as RR scheduling
- Starvation free as long as earlier jobs are bounded

Shortest Job First, SJF

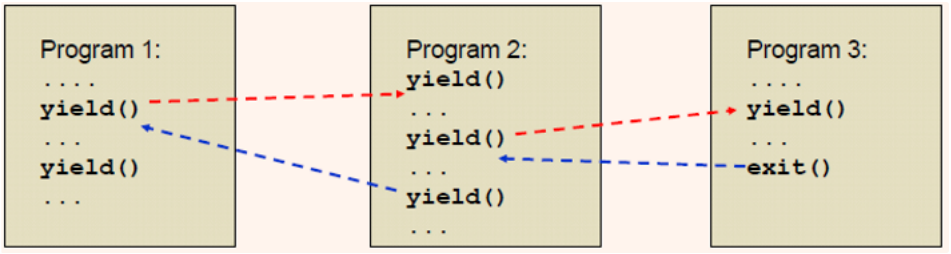
- Processes are ordered by total CPU time used
- Jobs that run for less time will run first
- Reduces average waiting time if number of processes is fixed
- Potential for starvation

> **Starvation:** 饥饿（starvation）指的是一个或多个可运行进程由于某种原因长时间得不到所需的资源，而无法进一步执行的情况。这种现象通常发生在并发控制或进程调度的环境中。

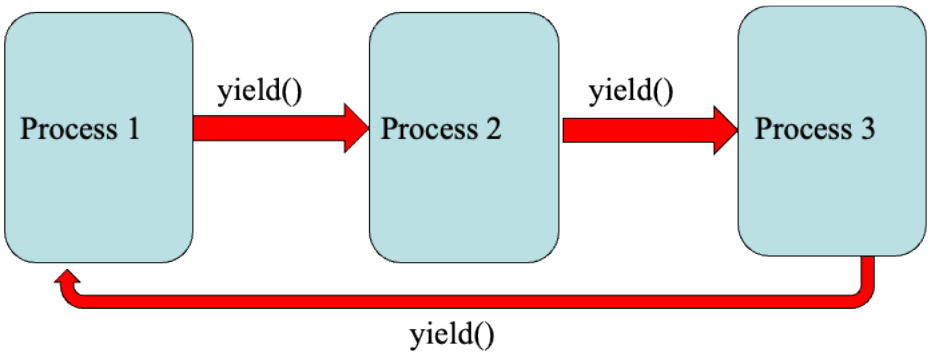
Co-operative Scheduling Policies

Round Robin for Voluntary Scheduling:

- Processes call a special "yield" function
  - This invokes the scheduler
  - Cause the process to be suspended and another process start up



- In many systems, Voluntary Scheduling is used with a round-robin arrangement



Pre-emptive Scheduling Policies

Shortest Remaining Time, SRT

- Pre-emptive form of SJF
- Processes are ordered according to remaining CPU time left

Round-Robin with Timer, RR

- Each process is given a fixed time slice c_i
- After time c_i, scheduler is invoked and next task is selected on a RR basis

13.5 Managing Multiple Policies

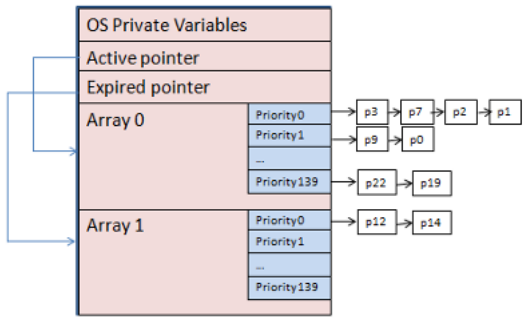
- Multiple policies can be implemented on the same machine using multiple queues:
  - Each queue can have its own policy
  - This scheme is used in Linux

high priority:	P1, P3	(RR policy)
medium priority:	P2	(RR policy)
low priority:	P4, P5	(batch queue, FCFS policy)

13.6 Scheduling in Linux

- Processes in Linux are dynamic:
  - New processes can be created with `fork()`
  - Existing processes can exit
- Priorities are also dynamic:
  - Users and superusers can change priorities using "nice" values
  - `nice -n 19 tar cvzf archive.tgz`
    - Allows tar to run with a priority lowered by 19 to reduce CPU load
    - Normal users can only set $0 \leq n \leq 19$
    - Superusers can specify $-20 \leq n \leq 19$. Negative nice increases priority
- Linux maintains three types of processes:
  - Real-time FIFO:
    - RT-FIFO processes cannot be pre-empted except by a higher priority RT-FIFO process.
  - Real-time Round-Robin:
    - Like RT-FIFO but processes are pre-empted after a time slice.
  - Linux only has “soft real-time” scheduling.
    - Cannot guarantee deadlines, unlike RMS and EDF we saw earlier.
    - Priority levels 0 to 99
  - Non-real time processes
    - Priority levels 100 to 139
- Linux maintains 280 queues in two sets of 140
  - An active set

- An expired set



- The scheduler is called at a rate of 1000Hz
  - Time tick is 1ms
  - RT-FIFO processes are always run if any are available
  - Otherwise:
    - Scheduler picks highest priority process in active set to run.
    - When its "time quantum" is expired, it is moved to the expired set. Next highest priority process is picked.
    - When active set is empty, active and expired pointers are swapped. Active set becomes expired set and vice versa.
    - Scheme ensures no starvation of lowest priority processes.
- What happened if a process becomes blocked? (For example, on I/O)
  - CPU time used so far is recorded. Process is moved to a queue of blocked processes
  - When process becomes runnable again, it continues running until its time quantum is expired
  - It is then moved to the expired set
- When a process becomes blocked its priority is often upgraded
- Time quantum for RR processes:
  - Varies by priority. For example:
    - Priority level 100 → 800ms
    - Priority level 139 → 5ms
    - System load
- How process priorities are calculated:
  - Priority = base + f(nice) + g(cpu usage estimate)
    - `f()` = priority adjustment from nice value
    - `g()` = Decay function. Processes that have already consumed a lot of CPU time are downgraded
  - Other heuristics are used:
    - Aging (age of process)
    - More priority for processes waiting for I/O - I/O boost
    - Bias towards foreground tasks

- I/O boost:
  - Rationale:
    - Tasks doing `read()` has been waiting for a long time. May need quick response when ready
    - Blocked/waiting processes have not run much
    - Applies also to interactive processes - blocked on keyboard/mouse input

Free Storage Space Management

- Similar to main memory management
- Linked list organisation
  - Linking **individual** blocks -- inefficient:
    - No block clustering to minimise seek operations
    - Groups of blocks are allocated/released one at a time
  - Better: Link groups of consecutive blocks
- Bit map organisation
  - Analogous to main memory
  - A single bit per block indicates if free or occupied

14 - Inter-Process Communication

14.1 Introduction

- In previous chapters, we looked at how multiple processes can run on a single CPU
- In real word applications, there are dependencies between processes
  - Process B cannot proceed because it is waiting for Process A's result
- In both process A and B are allowed to run freely, errors will occur
  - Process B proceeds before A completes, resulting in B using stale results
- Some form of co-ordination is therefore required

14.2 Race Conditions & Critical Sections

14.2.1 Race Conditions

- Race condition occur when two or more processes attempt to **access shared resources**:
  - Global variables
  - Memory locations
  - Hardware registers
  - CPU time
- Under this condition, the **un-predictable of execution order** for processes will cause the un-predictable of results and errors.
  两个或更多的进程或线程在访问共享资源时，它们的执行顺序导致不可预知的或错误的结果的情况。它通常发生在并发环境中，尤其是当多个操作必须以正确的顺序执行时，否则可能导致数据冲突。
  举例来说，假设有两个线程，它们都试图同时更新同一个变量。如果它们的操作没有适当地同步，一个线程的更新可能会覆盖另一个线程的更新，结果是变量中的数据不是任何一个线程预期的值。

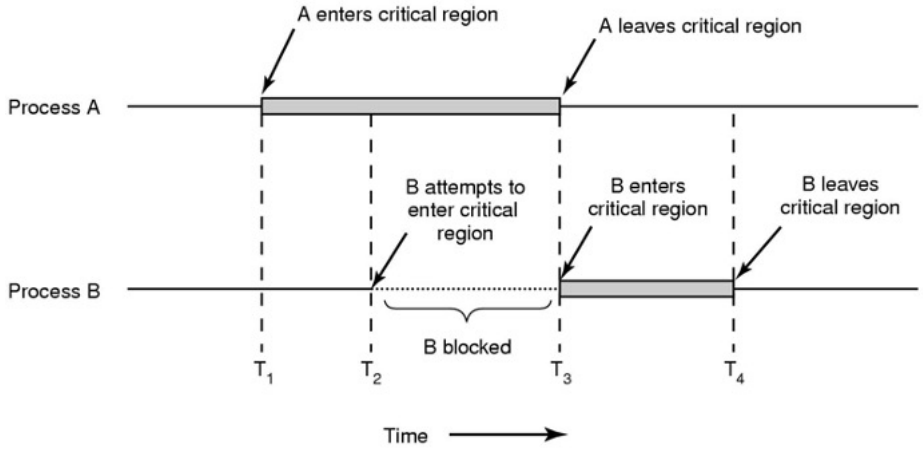
14.2.2 Critical Sections

- To prevent race conditions, we must prevent two processes from **reading/writing shared resources at the same time**
  - This is known as a "mutual exclusion" or "mutex" 互斥锁
- In concept, a running process is always in one of the two possible states:
  1. It is performing local computation. This does not involve global storage, hence race condition is not possible
  2. It is reading/updating global variable. This **can lead to race condition**
- When running process is in the **second state**, it is within its "**critical section**"
  我们将一段可以访问共享资源的代码范围定义为**临界区 Critical Section**。在这个范围内的代码会访问共享资源，如果多个进程的代码同时访问共享资源，则会出现race condition，这是我们需要避免的。
  所以我们先定义出一个代码区域或范围，其是会访问共享资源以出现潜在的race condition，以便进行下一步操作。

Mutual Exclusion

相互排斥（Mutual Exclusion）是指在并发编程中确保当一个线程进入临界区时，其他线程或进程必须等待，直到该临界区的线程退出，才能访问共享资源的一种原则或机制。

- To prevent race conditions, 4 rules must be followed:
  1. No two processes can in their critical section **at the same time**
  2. No assumptions may be made about speeds or numbers of CPUs
     - **Note**: we can relax this assumption for *most* embedded system since they have single CPU
     - May apply to systems using multi-core microcontrollers
  3. No process outside of its critical section can block other processes
  4. No process should **wait forever** to enter its critical section



14.3 Implementing Mutual Exclusion

- There are several ways of implementing mutexes, each with their own pros and cons:
  1. **禁用中断（Disabling interrupts）**：
     - 原理：在单处理器系统中，通过禁用中断，当前运行的代码可以防止上下文切换，从而避免进入临界区的竞争条件。
     - 优点：简单易实现；在临界区内的代码不会被打断。
     - 缺点：只适用于单处理器系统；增加了系统调用和中断响应的延迟；如果临界区内代码执行时间过长，可能影响系统响应性能。
  2. **锁变量（Lock variables）**：
     - 原理：使用一个共享变量作为锁，任何线程在进入临界区之前必须检查并设置这个变量的状态。
     - 优点：实现简单。
     - 缺点：可能引起忙等（busy waiting），浪费CPU资源；不满足原子操作，仍可能发生竞争条件。
  3. **严格轮换（Strict alternation）**：
     - 原理：严格按顺序轮换，每个线程轮流进入临界区。
     - 优点：简单，且保证了公平性。
     - 缺点：不是很灵活，因为即使一个线程不需要进入临界区，它也必须等待其轮次来临才能让其他线程进入。
  4. **Peterson's solution**:

    - 原理：是一种软件解决方案，结合了锁变量和严格轮换的概念，使得两个线程可以安全地交替进入临界区。
    - 优点：不需要特殊的硬件支持；实现了真正的相互排斥。
    - 缺点：仍然使用忙等；只适用于两个线程。
5. **测试并设置锁（Test-and-set lock）**：
    - 原理：使用一个原子操作的测试并设置（test-and-set）指令来实现锁。
    - 优点：是原子操作，因此在多处理器系统中也可以安全使用。
    - 缺点：可以导致忙等，尤其在锁争用较高的时候。
6. **睡眠/唤醒（Sleep/Wakeup）**：
    - 原理：使用操作系统提供的睡眠和唤醒调用来控制线程的执行，线程在不能进入临界区时会睡眠，在可以进入时被唤醒。
    - 优点：避免了忙等，CPU可以切换到其他任务。
    - 缺点：睡眠和唤醒操作的实现可能会导致额外的复杂性，如需要防止信号丢失或错误唤醒的情况。

14.3.1 Disabling Interrupts

- Time-slicing depends on a time interrupt. If interrupt is disable, the scheduler will never be activated to switch another process
- Similarly, process that are blocked pending an event, depend on an interrupt to tell the scheduler that the event has taken place
- Therefore, disabling interrupts will prevent switch to other process and enter to their critical section
- Cons:
    - Carelessly disabling interrupts can cause the entire system to grind to a halt
    - Only works on single-processor single core.

14.3.2 Lock Variables

- A single global variable "lock" is initially 1
- Process A reads this variable and set it to 0, and enter its critical section
- Process B reads this variable and see it's a 0. B does not enter its critical section and waits until "lock" is 1
- Process A finishes and set "lock" back to 1, allowing B to enter

Cons:

- The reading and updating of the global variable is not **atomic process**, still may cause race condition
    - **Atomic process原子操作**是指在执行过程中不会被其他任务或事件中断的操作原子操作的特点包括：
        1. **不可中断**：原子操作一旦开始，就会连续执行到完成，不会被其他线程或中断打断。
        2. **完整性**：它们要么完全执行，要么完全不执行，不会留下中间状态。
        3. **独占性**：在对共享资源（如内存位置）执行原子操作时，任何其他线程都不能访问该资源。
- **Busy waiting** occured, waste CPU resources

- **Busy waiting忙等**，又称为**自旋等待（Spinlock）**，是一种同步机制，其中一个进程或线程在等待某个条件变为真（如等待锁释放）时不断地检查这个条件，而不进行休眠或让出CPU给其他进程。这种等待方式中，进程或线程占用处理器时间进行无效的循环检查，而不是进行有用的工作。

14.3.3 Test and Set Lock (TSL)

Use command `TSL reg, lock;`, where `lock` is a variable in memory

- CPU locks the address and data buses, and reads "locks" from the memory
  - The locked address and data buses will block access from all other CPUs
- The current value is written into register "reg"
- A "1" value is written into `lock`
- CPU unlocks the address and data buses
- It is "atomic", means that nothing can interrupt execution of this instruction, which is guaranteed in hardware.

```
enter_region:
    TSL REGISTER,LOCK          | copy lock to register and set lock to 1
    CMP REGISTER,#0            | was lock zero?
    JNE enter_region           | if it was non zero, lock was set, so loop
    RET | return to caller; critical region entered


leave_region:
    MOVE LOCK,#0               | store a 0 in lock
    RET | return to caller
```

14.3.4 Sleep/Wakeup

- The solution of the 'busy wait' is through the use of "sleep/wake" function
  - When a process finds that a lock has been set, it calls `sleep()` function and put itself into the blocked state
  - When the other process exits the critical section and clears the lock, it calls `wake()` function which moves *all* the blocked process into the READY queue
- This approach can create a problem called "producer-consumer problem"

14.4 The Producer/Consumer Problem

生产者-消费者问题（Producer-Consumer Problem）是一个经典的并发问题，涉及两类进程、线程或实体：生产者（Producers）和消费者（Consumers）。生产者负责生成数据、工作项、任务等，而消费者则负责处理生产者生成的这些项。这两类实体必须同步操作，以便生产者不会在消费者处理完当前数据之前覆盖数据，同时消费者在没有数据可处理时不会进行无效操作。

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count - 1;                    /* decrement count of items in buffer */
          if (count == N - 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

- Producer and consumer share a fixed-size buffer
  - A global variable `count` keeps track of the number of items
    - If `count == N (full)`, producer sleeps, if `count == 0 (empty)`, consumer sleeps
  - After reading from the buffer, check the buffer size:
    - if consumer check `count == N-1 (not full)`, wake up producer
    - if producer check `count == 1 (not empty)`, wake up consumer
- Potential deadlock:
  - Consumer costs the last item in the buffer, `count` now is 0
  - Consumer wakeup producer since count == N-1, consumer start consuming items but **NOT SLEEP**
  - Producer wakes and add one item to buffer, increase `count` to 1, then wake up the consumer. However, the consumer now is still consuming the item and **NOT SLEEP**, the `wake()` is lost.
  - Consumer finished consuming and start-over the while loop, check the count is 0, consumer **SLEEP**
  - Producer get up and produce items until the buffer is full, producer **SLEEP**
  - No one is awake, deadlock

14.5 Semaphores

- A semaphore is a special lock variable that counts the number of wake-ups saved for future use
  - A value of '0' indicates that no wake-ups have been saved
- Two atomic operations on semaphore
  - DOWN, TAKE, PEND or P:
    - If the semaphore has a value > 0, it is decremented and the DOWN operation returns
    - If the semaphore is 0, the DOWN operation blocks

- UP, POST, GIVE or V
  - If there are any processes blocking on a DOWN, one is selected and waken up
  - Otherwise UP increments the semaphore and returns
- **等待（P）**：线程在尝试进入临界区之前调用这个操作。如果信号量的值大于零，信号量的值减一，线程进入临界区。如果信号量的值已经是零，这意味着没有可用的资源，线程将被阻塞，直到信号量的值变为正。
- **发信号（V）**：线程在离开临界区时调用这个操作。信号量的值增加一，如果有线程正在等待这个信号量，则其中一个将被唤醒。

```
#define N 100                           /* number of slots in the buffer */
typedef int semaphore;                  /* semaphores are a special kind of int */
semaphore mutex = 1;                    /* controls access to critical region */
semaphore empty = N;                    /* counts empty buffer slots */
semaphore full = 0;                     /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                     /* TRUE is the constant 1 */
          item = produce_item( );       /* generate something to put in buffer */
          down(&empty);                 /* decrement empty count */
          down(&mutex);                 /* enter critical region */
          insert_item(item);            /* put new item in buffer */
          up(&mutex);                   /* leave critical region */
          up(&full);                    /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                     /* infinite loop */
          down(&full);                  /* decrement full count */
          down(&mutex);                 /* enter critical region */
          item = remove_item( );        /* take item from buffer */
          up(&mutex);                   /* leave critical region */
          up(&empty);                   /* increment count of empty slots */
          consume_item(item);           /* do something with the item */
     }
}
```

Mutual Exclusion with Semaphore

- When a semaphore's counting ability is not needed, we can use a simplified version called "mutex"
  - 1 = Unlocked
  - 0 = Locked
- Two processes can then attempt do DOWN the semaphore
  - Only one will succeed. The other will block

- When the successful process exits the critical section, it does an UP to wake up others

Process A	Process B
sema=1	
...	
non_critical_section()	non_critical_section()
DOWN(sema)	DOWN(sema)
critical_section()	critical_section()
UP(sema)	UP(sema)
...	...

14.6 Deadlocks with Semaphores

- Our producer/consumer solution swapped the semaphores for empty/full with the mutex semaphore, the potential deadlock occurs

```
#define N 100                           /* number of slots in the buffer */
typedef int semaphore;                  /* semaphores are a special kind of int */
semaphore mutex = 1;                    /* controls access to critical region */
semaphore empty = N;                    /* counts empty buffer slots */
semaphore full = 0;                     /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                     /* TRUE is the constant 1 */
          item = produce_item();        /* generate something to put in buffer */
          down(&mutex);                 /* decrement empty count */
          down(&empty);                 /* enter critical region */
          insert_item(item);            /* put new item in buffer */
          up(&mutex);                   /* leave critical region */
          up(&full);                    /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                     /* infinite loop */
          down(&mutex);                 /* decrement full count */
          down(&full);                  /* enter critical region */
          item = remove_item();         /* take item from buffer */
          up(&mutex);                   /* leave critical region */
          up(&empty);                   /* increment count of empty slots */
          consume_item(item);           /* do something with the item */
     }
}
```

- When producer successfully DOWN the mutex, it go to he next code for check the empty semaphore
- The empty semaphore has value 10, which is full, producer blocked
- Consumer check mutex, which is DOWNed by producer, consumer blocked
- Deadlock

Deadlock: Reusable/Consumable Resources

- Reusable Resources:
  - Memory, devices, files, tables...
  - Number of units is **constant**

    - Unit is either free or allocated; **no sharing**
    - Process **requests, acquires, releases units**
- Consumable Resources
    - Messages, signals, ...
    - Number of units **varies** at runtime
    - Process releases (create) units (without acquire)
    - Other process **requests** and **acquires** (consumes)
      可重用资源（Reusable Resources）：

- **定义**：可重用资源是指在系统中数量固定，可以被多个进程共享使用的资源。这类资源使用完毕后不会消失，可以被释放并重新分配给其他进程。
- **特点**：
    - **数量固定**：如内存、设备、文件和数据库表等，它们的总数在运行时不会改变。
    - **非共享**：在任一时刻，每个单元要么是空闲的，要么被某个进程独占。
    - **请求-获取-释放**：进程使用这些资源时通常遵循请求资源、获取资源、最终释放资源的周期。

可消耗资源（Consumable Resources）：

- **定义**：可消耗资源是指它们的数量会随着系统的运行而变化，通常是由进程创建，并被其他进程消耗。
- **特点**：
    - **数量变化**：如消息或信号等，它们可以在运行时被创建和消耗，因此它们的总数是可变的。
    - **创建和消费**：一个进程可以释放（或说是创建）资源，无需先获取资源。而另一个进程可能会请求和获取（消费）这些资源。

死锁的关联：

- **可重用资源死锁**：如果多个进程各自持有一部分资源，并请求更多的资源时，可能导致循环等待的情况，这是死锁的经典场景。
- **可消耗资源死锁**：尽管可消耗资源不像可重用资源那样直观地与死锁关联，但如果进程间的信号通信不当，也可能导致死锁。例如，一个进程等待从另一个进程接收信号，而后者也在等待某种资源或信号，这可能形成死锁。

Dealing with Deadlocks

1. Detection and Recovery
    - Allow deadlock to happen and eliminate it
2. Avoidance (dynamic)
    - Runtime checks disallow allocations that might lead to deadlocks
3. Prevention (static)
    - Restrict type of request and acquisition to make deadlock impossible
      处理死锁的策略：
4. **检测和恢复**：
    - 允许死锁发生，但需要有机制来检测它，并一旦检测到就采取措施恢复系统，比如中断并重启涉及的进程。
5. **避免（动态）**：
    - 在运行时进行检查，以阻止可能导致死锁的资源分配。这涉及到对资源分配请求进行评估，以确保它们不会引起系统的不安全状态。

6. **预防（静态）**：
    - 通过限制请求和分配资源的方式，从根本上排除死锁的可能性。

Deadlock Prevention

- Deadlock requires the following 3 conditions:
    1. **Mutual exclusion**: resources not sharable
    2. **Hold and wait**: process must be holding at least one resource while request another
    3. **Circular wait**: at least 2 processes must be blocked on each other
       死锁通常需要以下三个条件同时满足：

1. **互斥**：资源不能共享，必须由一个进程独占。
2. **保持并等待**：进程至少持有一个资源，并且正在等待获取额外的资源。
3. **循环等待**：存在一个进程链，每个进程都在等待下一个进程持有的资源。

Eliminate mutual exclusion

- Not possible in most cases. 在大多数情况下，这是不可能的，因为某些资源（如打印机）本质上就是不可共享的。
- Spooling makes I/O device sharable. 通过技术如假脱机（Spooling）可以使某些I/O设备变得可共享。

Eliminate hold-and-wait

- Request *all resources* at once. 要求进程一次性请求其需要的所有资源
- Release *all resources* before a new request. 要求进程一次性请求其需要的所有资源
- Release *all resources* if current request blocks. 如果当前请求无法立即满足，则释放所有已持有的资源

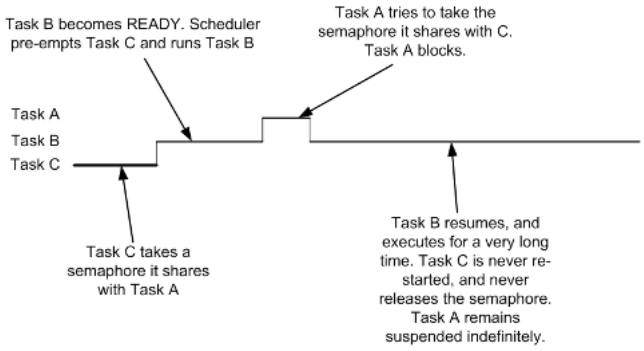
Eliminate circular wait

- Order all resources. 对系统中的所有资源进行排序
- Process must request in ascending order. 要求每个进程必须按照资源编号的升序来请求资源

Problem with Semaphores: Priority Inversion

优先级反转（Priority Inversion）是操作系统中的一个经典问题，发生在一个高优先级任务被迫等待一个低优先级任务释放资源的情况。这通常是因为有一个中等优先级的任务阻止了低优先级任务的执行，而高优先级任务又在等待低优先级任务持有的资源。

- In the diagram below, priority (Process C) < priority (Process B) < priority (Process A)



- Process B effectively blocks out Process A, although Process A has higher priority

1. 低优先级任务C开始执行，并获得了一个信号量（或其他同步资源）。
2. 在任务C完成操作并释放信号量之前，一个中等优先级任务B开始执行，并且由于调度策略，它抢占了任务C的CPU时间。
3. 高优先级任务A开始执行，并需要之前被任务A获得的那个信号量。然而，由于任务C还没有释放信号量，任务A不能继续，即便它有更高的优先级。
4. 由于任务B持续占用CPU（因为它优先级高于任务A），任务C无法运行，因此也就无法释放信号量。这样，高优先级的任务A被迫等待低优先级的任务C，而任务C又因为任务B而无法运行。
   Out of context:
   优先级反转的问题在于，它违反了优先级调度的基本原则：高优先级任务应该被优先执行。在优先级反转的情况下，一个低优先级任务可能会无意中阻塞一个高优先级任务的执行，这可以导致性能下降，甚至在严重的实时系统中可能导致系统失败。
   为了解决优先级反转问题，可以采用几种策略：

- **优先级继承（Priority Inheritance）**：如果一个低优先级任务持有一个高优先级任务需要的资源，那么低优先级任务临时继承高优先级任务的优先级，直到它释放该资源。
- **优先级天花板（Priority Ceiling）**：系统中每个信号量都有一个预先定义的“天花板”优先级，任何持有该信号量的任务都将运行在这个优先级，以防止更低优先级任务的干预。
- **队列管理**：更改调度队列的管理，确保高优先级任务优先得到服务。

14.7 Monitors & Conditional Variables

14.7.1 Monitors

- A monitor is similar to a class or abstract-data type in C++ or Java:
  - Collection of procedures, variables and data structures grouped together in a package
    - Access to variables and data possible only though methods defined in the monitor
  - However, only one process can be active in a monitor at any point of time
    - I.e., if any other process tries to call a method within the monitor, it will block until the other process has exited the monitor
- Implementation:
  - When a process calls a monitor method, the method first checks to see if any other process is already using it.

    - If so, the calling process blocks until the other process has exited the monitor
      监视器Monitor：
- 这是一种同步构造，其封装了资源共享的访问，提供了一种安全地允许多个进程访问同一资源的方式。其他任何试图访问监视器的任何方法methods的进程都会被阻塞，直到当前的method执行完成。

14.7.2 Monitors and Condition Variables

- Monitors achieve mutual exclusion, but we also need other mechanisms for coordination
    - E.g. in our producer/consumer problem, mutual exclusion is not enough to prevent the producer from proceeding when the buffer is full
- We introduce "**condition variable**"
    - One process WAITs on a condition variable and blocks, until...
    - Another process SIGNALs on the same condition variable, unblocking the WAITing process
      条件变量：监视器使用条件变量来挂起和唤醒线程。这些条件变量是监视器对象的一部分，允许线程在某些条件不满足时等待（wait），并在条件可能已变为真时被唤醒（signal）。
- Implementing the Producer/Consumer Problem with semaphores and condition variables:
    - When the buffer is full (`count == N`), producer will WAIT on a full condition
    - When the buffer is empty (`count == 0`), consumer will WAIT on empty.

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count - 1;
        if count = N - 1 then signal(full)
    end;
    count := 0;
end monitor;

procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

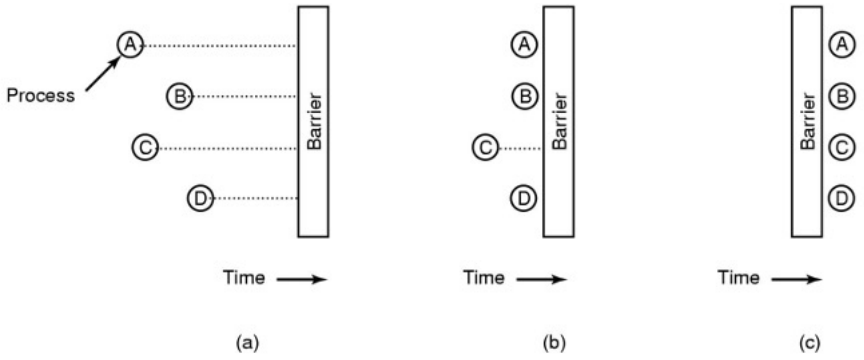
- When a process encounters a WAIT, it is blocked and another process is allowed to enter the monitor
- Problem:
    - When there's a SIGNAL, the sleeping process is woken up
    - We will potentially now have two processes in the monitor at the same time:
        - The process doing the SIGNAL
        - The process that woke up because of the SIGNAL
- We have 3 ways to resolve this:

1. We require that the signaler exits immediately after calling SIGNAL
2. We suspend the signaler immediately and resume the signaled process
3. We suspend the signaled process until the signaler exits, and resume the signaled process only after that

- A condition variable is different from a semaphore.
  - Semaphore:
    - If Process A UPs a semaphore with no pending DOWN, the UP is **saved.**
    - The next DOWN operation will not block because it will match immediately with a preceding UP.
  - Condition variable:
    - If Process A SIGNALs a condition variable with no pending WAIT, the SIGNAL is simply **lost.**
    - This is similar to the SLEEP/WAKE problem earlier on.

14.8 Barriers

- A "barrier" is a special form of synchronisation mechanism that works with groups of processes rather than single processes
  屏障(Barrier) 是一种特殊形式的同步机制，用于协调一组进程或线程，而不是单个进程或线程。它通常用于并行编程和多线程应用中，确保在某个执行点上所有的进程或线程都达到了屏障点，然后才能一起继续执行。换句话说，它是一种集体等待点，直到所有成员都准备好了，才能跨过这个点。
  如何工作：
  当一个进程或线程到达屏障点时，它会在那里等待，直到所有其他进程或线程也都到达这一点。一旦最后一个进程到达屏障点，所有在屏障点等待的进程或线程就可以继续执行。
  使用场景：
  屏障在以下情况下特别有用：
- **并行计算**：在数据处理或计算密集型任务中，可能需要将任务分割成多个部分并行处理。屏障可以确保各个部分在继续下一步之前都完成了当前步骤。
- **同步启动**：确保所有线程或进程都已准备好，然后同时开始执行。
- **迭代算法**：在每个迭代步骤结束时同步，比如在使用迭代方法求解数值问题时。



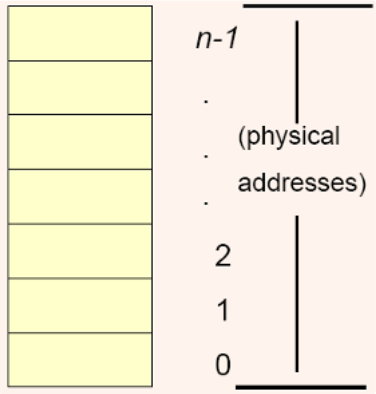
15 - Memory Management

15.1 Introduction

- Memory is crucial for computers:
  - Used to store:
    - Kernel code and data
    - User code and data
- What responsibilities does the OS have here?
  - Allocate memory to new processes
  - Manage process memory
  - Manage kernel memory for its own use
  - Provide OS service to:
    - Get more memory, e.g. via `malloc`
    - Free memory, e.g. via `free`
    - Protect memory

15.2 Physical Memory Organisation

- Physical memory is:
  - The actual matrix of capacitors (DRAM) or flip-flops (SRAM) that stores data and instructions
  - Arranged as an array of bytes
- **Memory addresses serve as byte indices**



Words

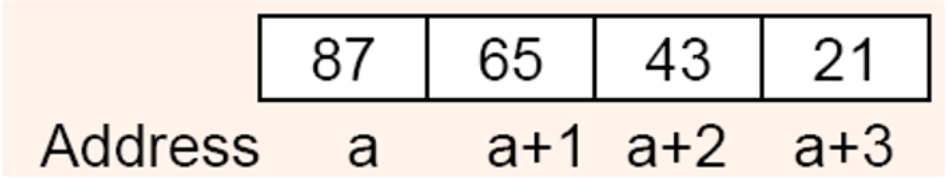
- Physical memory is organised in bytes, but CPU often transfer data in units of >1 byte
  - This unit is called **word**

- For 8-bit machine, 1 word = 1 byte,
  For 16-bit machine, 1 word = 2 bytes,
  For 32-bit machine, 1 word = 4 bytes
  For 64-bit machine, 1 word = 8 bytes

Endianness (Big Endian & Little Endian)

How to store multiple byte word (> 1 byte), 2 general schemes, e.g. `0x87654321`

- **Big Endian:** higher order bytes at lower address
  在大端字节序中，最高有效字节（MSB）存储在最低的内存地址上，而最低有效字节（LSB）存储在最高的内存地址上。这类似于我们阅读数字的方式，从左到右读，高位在前。

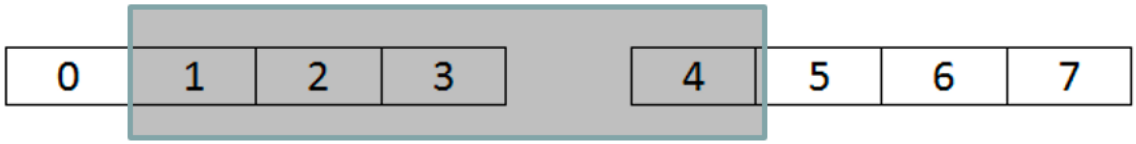


- **Little Endian:** lower order bytes at lower
  在小端字节序中，最低有效字节（LSB）存储在最低的内存地址上，而最高有效字节（MSB）存储在最高的内存地址上。这类似于我们阅读数字的方式倒过来，低位在前。

  x86: Little Endian, TPC/IP: Big

Alignment

- Data can be fetched across word boundaries

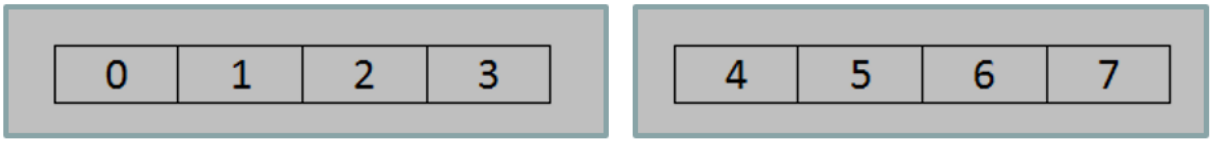


- E.g. fetching from address 1 in a 32-bit machine
  - Bytes from address 0 to 3 are fetched
  - Bytes from address 4 to 7 are fetched
  - Bytes from address 0, 5, 7 are discarded
  - Bytes from 1, 2, 3, 4 are merged
- 这种没有对齐的数据依然可以被正常读取，但是相比于对齐的数据，在这个例子中未对齐的数据传输了多一倍的数据
- Since mis-aligned reads are very inefficient, data structures should always be created in units of words

- E.g., add unused bytes to ensure this

```
eg. struct {
      char a;
      int b; // 32 bits, must be aligned on 32-bit boundary
    } x;
treat as if padding added
struct {
      char a;
      char pad[3]; // force address of b to be aligned
      int b; // b now aligned
    } x1;

sizeof(x1) = 8; if x1 is aligned on its size (8 bytes) then b is also aligned
  on 32-bit boundary
```

- Due to CPU fetch circuitry design, instruction usually must be fetched on word boundaries



- Instructions fetched across word boundaries trigger "**Bus Error**" faults

15.3 Memory Management

- Why Memory Management?
  - We want to use memory **efficiently**
    - What memory has already been allocated to whom?
    - What memory is now free and usable for others?
  - We want to **protect** process from each other
    - One process should not to able to trash another process or the OS
    - E.g., we don't want Process 1 to be messing about with the variables and memories used by Process 2

Logical & Physical Addresses

- **Logical addresses**:
  - These are addresses as "seen" by executing processes code
  - 也被称为虚拟地址（Virtual Address），是由执行程序产生的地址。
  - 对于程序中的任何内存请求，都会生成逻辑地址。
  - 逻辑地址是**相对地址**，它是从**程序开始的地方**计算的。
  - 它由程序的代码通过内存管理单元（Memory Management Unit，MMU）在运行时转换为物理地址。

    - 逻辑地址使得程序员无需关心内存中的实际物理位置，可以使用一致的地址空间来编写程序。
- **Physical addresses:**
    - These are addresses that are *actually* sent to memory to retrieve data or instructions
    - 物理地址是数据实际存储在物理内存中的地址。
    - 它是由计算机的硬件和操作系统管理的。
    - 当MMU转换逻辑地址时，它会使用页表或其他数据结构来找到对应的物理地址。
    - 物理地址直接指向内存中的一个实际位置。

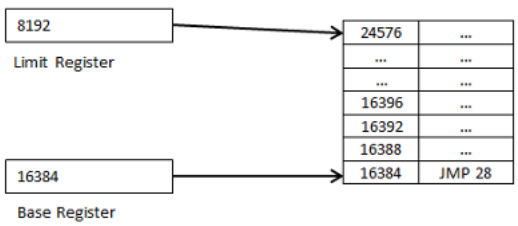
Multiple Program Systems

多程序系统是一种操作系统环境，它允许多个程序同时驻留在内存中，以便CPU可以在它们之间切换来实现多任务处理。这种系统的设计旨在提高资源利用率和系统吞吐量，因为当一个程序等待某些事件（如I/O操作）完成时，CPU可以切换到另一个程序继续工作，从而减少CPU空闲时间。
在多程序系统中，操作系统负责内存管理，确保每个程序有独立的地址空间，防止程序之间相互干扰。这通常通过使用逻辑地址和物理地址的转换来实现，即程序编写时使用逻辑地址，运行时操作系统和硬件负责将逻辑地址映射到物理内存地址。

- Having multiple processes complicates memory management:
    - **Conflicting addresses:** >1 program expects to load at the same place in memory
    - **Access violations:** program overwrites the code/data of another program/OS
    - The ideal situation would be to give each program a section of memory to work with
        - Basically each program will have its own address space

Base and Limit Registers

- Base Register:
    - This contains the **starting address** for the program
    - All program address are computed relative to this register
    - 存储分配给程序的物理内存块的起始物理地址。当程序中的逻辑地址生成时，系统会将逻辑地址加上基址寄存器的值来得到物理地址。
- Limit Register:
    - This contains the **length of the memory segment**
    - 存储程序分配的内存块的大小。它用来检查生成的物理地址是否在分配的内存块内。
- These registers solve both problems:
    - We can resolve **address conflicts** by setting different values in the base register
    - If a program tries to **access memory below** the base register value or **above** the (base + limit) register value, a "**segmentation fault**" occurs



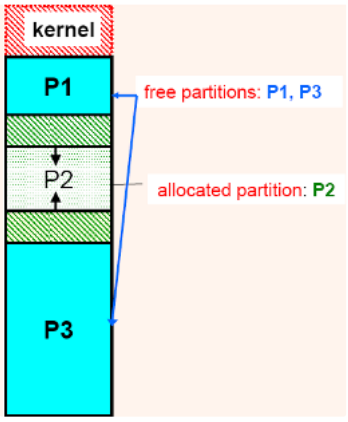
- All memory references in the program are relative to the Base Register
  - For example, `JMP 28` will cause a jump to memory location `16384+28=16412`
- Any memory access to location `16384+8192=24576` and above (or `16383` and below) will cause **segmentation faults**
  - Other programs will occupy spaces above and below the segment given

Partitioning

- Base and limit registers allow us to partition memory for each running process
  - Each process has its own **fixed partition**
  - Assumed that we know how much memory each process needs

Partitioning Issues: Fragmentation

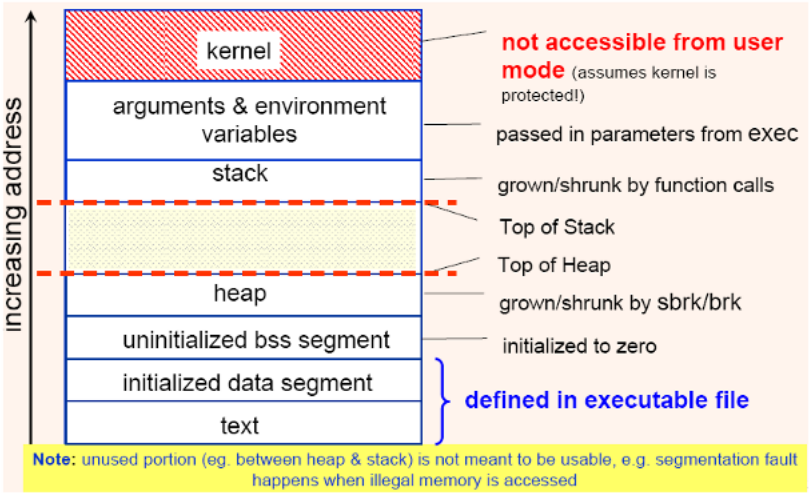
- Two types of fragmentation can arise
  - **Internal fragmentation:**
    - Partition is much larger than is needed
    - Cannot be used by other processes
    - Extra space is wasted
  - **External fragmentation:**
    - Free memory is broken into *small chunks* by allocated memory
    - Sufficient free memory in TOTAL, but individual chunks insufficient to fulfils requests
- 即外部碎片化和内部碎片化
  - 内部碎片化指程序被分配到过大的内存空间，剩余的内存无法被其他程序使用，造成浪费
  - 外部碎片化指剩余内存被分为很多个小片，剩余内存的总空间足够分配给其他程序，但是单个内存切片不能满足程序需求



Manage Memory within Processes

- We have seen how we can manage multiple processes within an OS, but:
  - What about memory within individual programs?
    - OS allocates memory for instructions

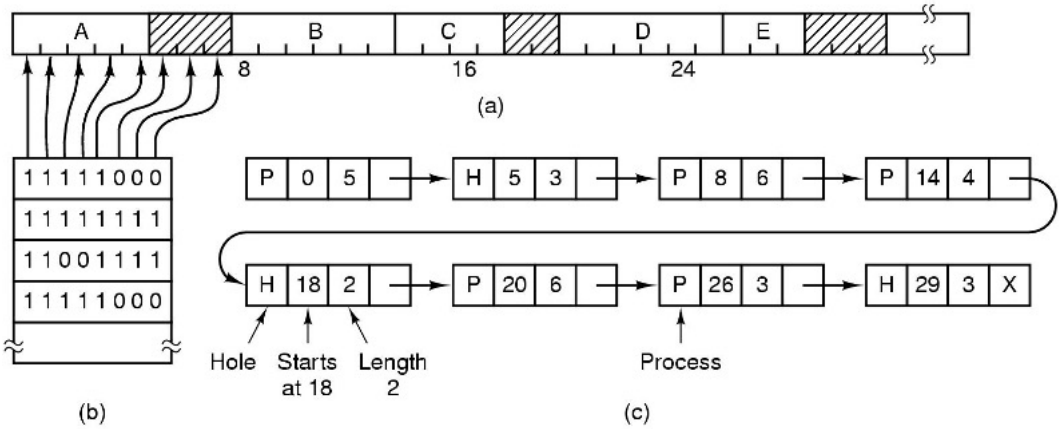
    - Global variables are created as part of the program's environment, and don't need to be specially managed
    - A "stack" is used to create local variables and store local addresses
        - LIFO, allocate space when a function is called, release space when function returned
    - A "heap" is used to create dynamic variables
        - Dynamically arranged
- E.g. In UNIX, process space is divided into:
    - **Text segments**: Read-Only, contains code. May have > 1 text segments
    - **Initialised Data**: Global data initialised from executable file. `char *msg[] = "Hello World!"`
    - **BSS Segment**: Contains uninitialised globals
    - **Stack**: Contains statically allocated local variables and arguments to functions, as well as return addresses
    - **Heap**: Contains dynamically allocated memory



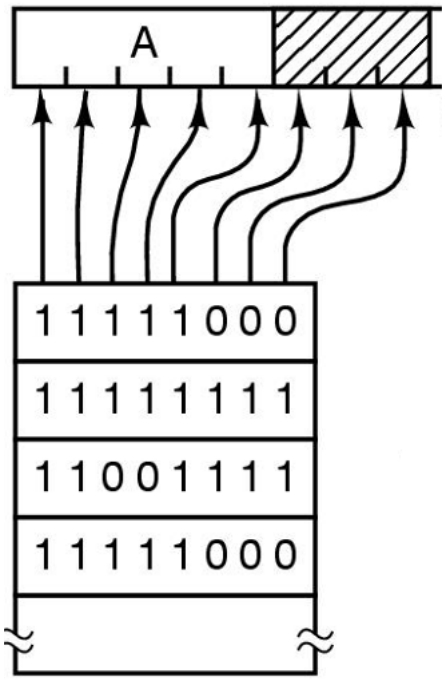
Managing Free Memory

- When memory can be allocated and de-allocated (e.g. using `malloc` / `free` or `new` / `delete`), it has to be managed by the OS
    - E.g., when a program requests for 168 Bytes of memory, where should it comes from?
    - When a program free 215 Bytes of memory, what should happen?
    - Should small fragments of free memory scattered all over be compacted?
- To manage free memory, we must know where these free chunks of memory are.
- Two approaches:
    - Bit maps
    - Free/Allocated List
- In either approach, memory is divided up into fixed sized chunks called "**allocation units**"
    - Common sizes range from several bytes (e.g. 16 bytes) to several kilo-bytes

- Each "tick mark" in figure (a) represents the boundary of an allocation unit



Bit Maps



(b)

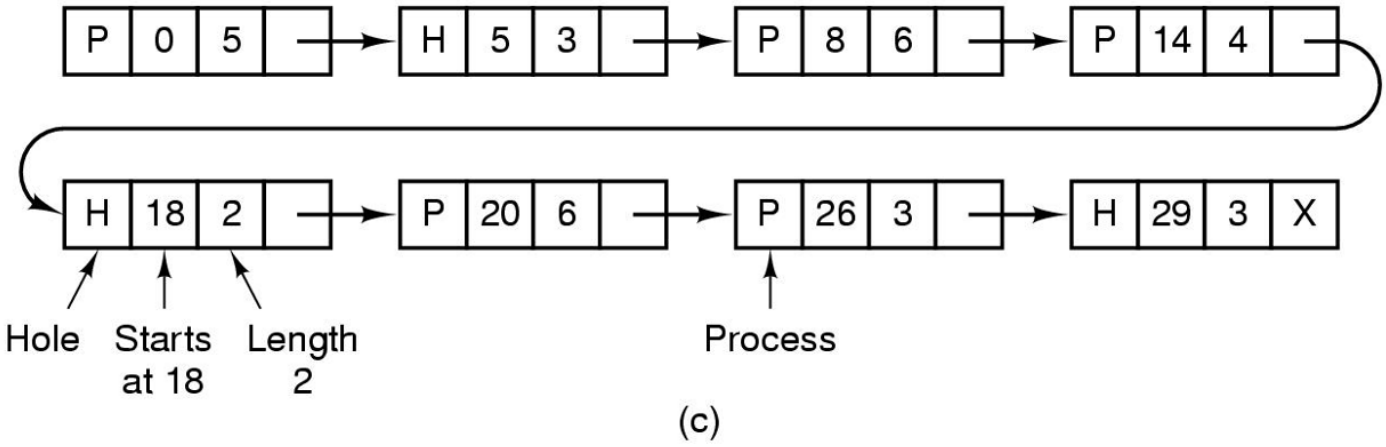
- Figure (b) shows how **bit maps** are used to keep track of free memory。
  - Each bit corresponds to an allocation unit
    - 0 indicates a free unit, 1 indicates an allocated unit
  - If a program request for 128 bytes:
    - Find how many allocation units are needed. if each unit if 16 bytes, this corresponds to 8 units
    - Scan through the list to find 8 **consecutive** 0's
    - Allocate the memory found, and change 0's to 1's
  - If a program frees 64 bytes:

    - Mark the bits corresponding to the 4 allocation units as 0

      位图（Bit Map）：
      位图是一种数据结构，用于跟踪内存块的使用情况。在这个结构中，内存被分成固定大小的块，每个块由位图中的一个位（bit）表示。如果一个位设置为0，它表示相应的内存块是空闲的；如果设置为1，则表示内存块被占用。

- **优点**：
  - 位图在表示大量内存块时非常高效，因为每个内存块只需要一个位来表示其状态。
  - 位操作通常非常快速，尤其是在现代计算机系统上。
- **缺点**：
  - 位图的缺点是它不够灵活，每个块的大小是固定的，这可能导致内存碎片问题，特别是如果块大小和实际请求不匹配时。
  - 对于非常大的内存区域，位图自身可能也会占用相当多的内存空间。

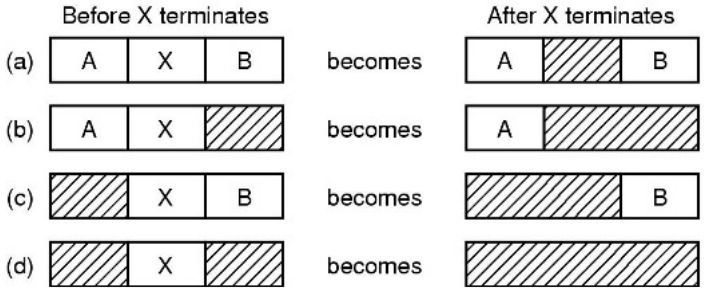
Free/Allocated List



(c)

- Figure (c) shows an alternative method:
  - A **single linked list** is used to track **allocated ("P")** units and **free ("H")** units
    - Each node on the linked list also maintains where the block of free units start, and how many consecutive free units are present in that block
  - Allocating free memory becomes simple:
    - Scan the list until we reach a **"H"** node that points to a block of a sufficient number of free units
- Can also implemented as a **doubly linked list**
  - Diagram below shows the possible "neighbour combinations" that can occur when a process X terminates

- The "back pointer" in doubly linked list makes it easy to combine freed blocks together



已分配链表 (Allocated List):
已分配链表是另一种内存管理技术，它使用链表数据结构来跟踪空闲内存块。链表中的每个节点代表一个空闲内存块，包含块的起始地址和大小。

- **优点**:
  - 已分配链表允许更加灵活的内存分配，因为它可以精确地跟踪每个空闲块的大小，所以可以根据需要分配不同大小的内存块。
  - 它可以通过合并相邻的空闲块来减少内存碎片。
- **缺点**:
  - 搜索合适的空闲块可能比较慢，尤其是当空闲列表很长时，可能需要遍历整个列表来找到足够大的空闲块。
  - 空闲列表可能需要额外的存储空间来维护链表结构。

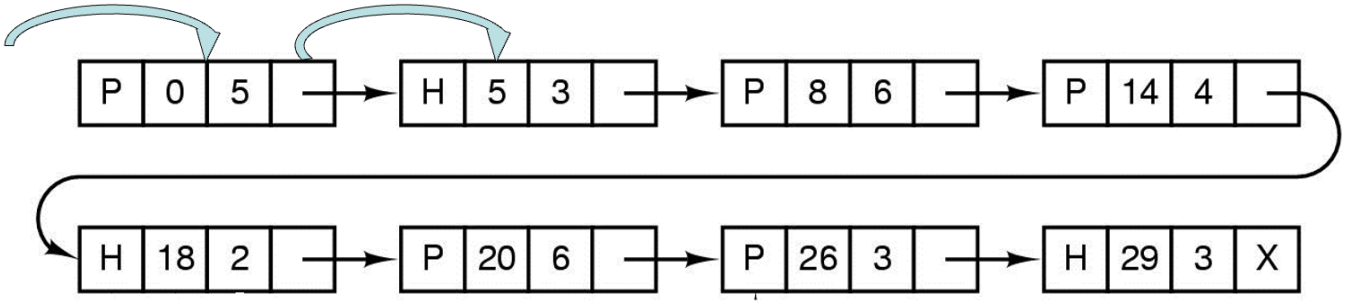
Allocation Policies

- The bitmap and allocated list tell us where the free space is, but not allocate
- There are several possibilities on how to do allocation:
  - Note that in each case, the free allocation units which are left at the end of an allocated block will returned to the free list
    - E.g. if a block of 8 units is found and only 6 are needed, the remaining 2 units are marked as "free"
  - **First Fit**
    - Scan through the list/bit map and find the **first block** of free units that can **fit the requested size**
    - Fast, easy to implement
  - **Best Fit**
    - Scan through the list/bit map to find the **smallest block** of free units that can **fit the requested size**
    - Minimise waste
    - It can lead to scattered bits of tiny useless holes
  - **Worst Fit:**
    - Find the largest block of free memory
    - Can reduce the number of tiny useless holes
- We can sort the free memory from the smallest to largest for **best fit**, or largest to smallest for **worst fit**

- This minimise the search time
- However combining free neighbours will becomes much harder

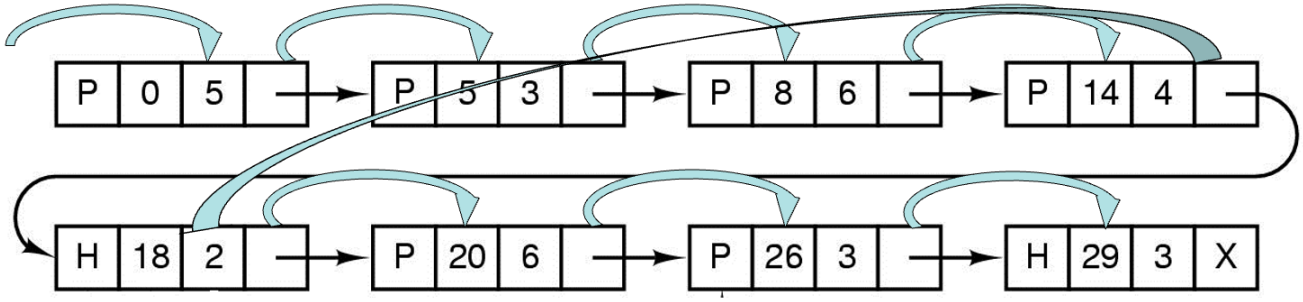
First Fit

- First allocation: 3 units



First Allocation

- Second allocation: 3 units

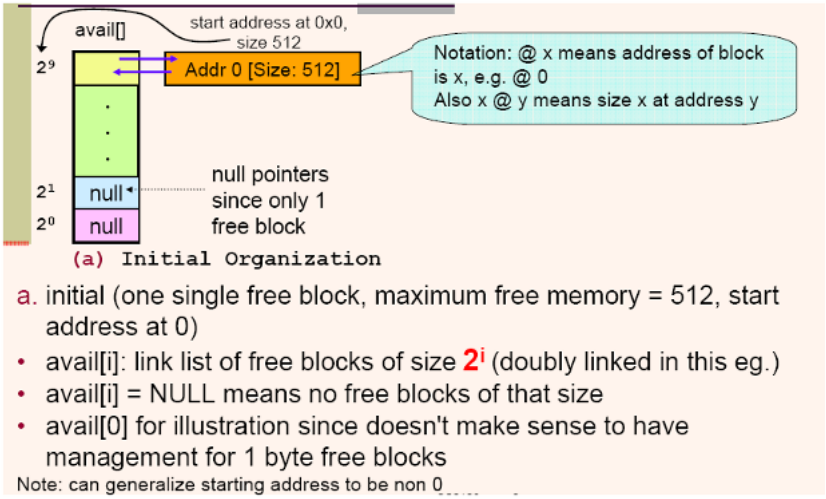


Second Allocation

Buddy Allocation (Quick Fit)

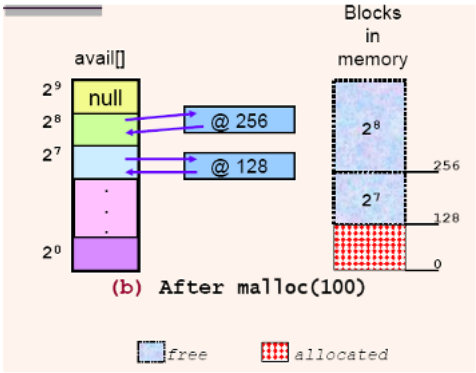
- This is an efficient (better than $O(n)$) way to manage free blocks
  - Binary splitting
    - Half of the block is allocated
    - The two halve are called "buddy blocks"

- Can combine again when two buddy blocks are free



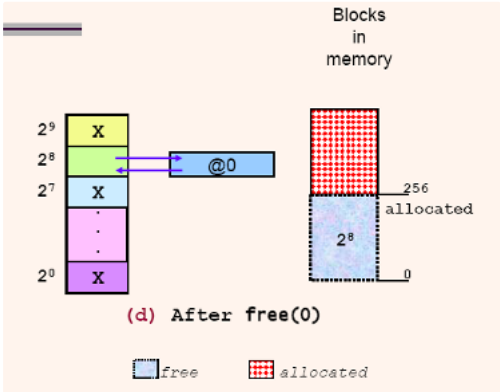
Example:

- We initially have a free block of 512 bytes
- We want to do `malloc(100)`, allocate 100 bytes
- Steps:
  1. **Split** 512 byte block into 2 sub-blocks of **256 bytes**
  2. **Split** one 256 bytes block into 2 sub-blocks of **128 bytes**
  3. Allocate memory location **0-127**
- These allocations have created blocks at addresses:
  - **0 to 127**: the block given to the malloc, since we return the first free block of at least 100 bytes.
  - **128-255**: The free buddy block of the block at address 0.
  - **256-511**: The free buddy block of the block that was broken up to 0-127 and 128-255.



- Now if we want to allocate 256 bytes
  - Block 256 is returned
- Now we call `free(0)`, free memory 0-127
  - This frees up addresses 0 to 127

- Coalesces (Combine) with is buddy block, 128-255, to form a single 256 byte block at address 0



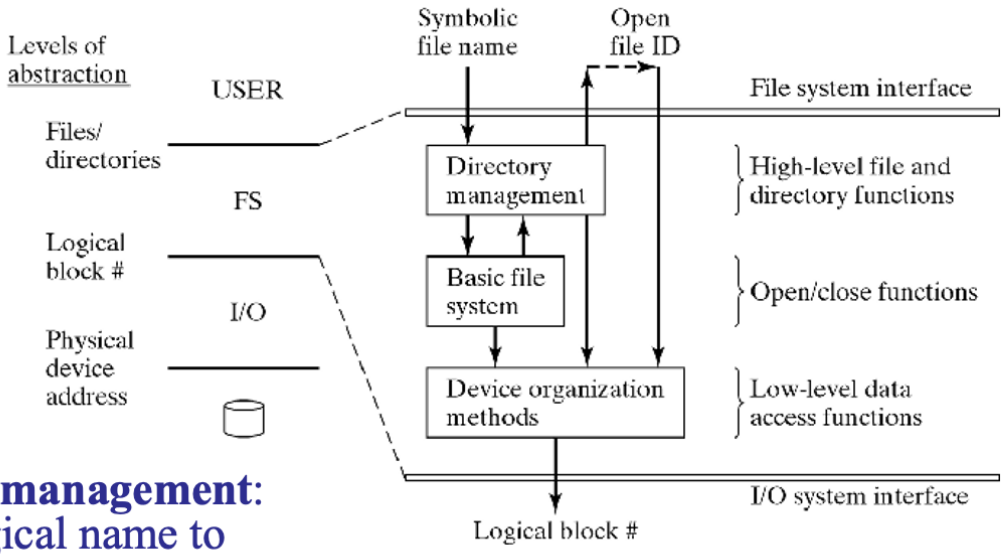
17 - File Systems

17.1 Introduction

What is a File System?

- Present logical (abstract) view of files and dictionaries
  - Accessing a disk is very complicated:
    - 2D or 3D structure, track/surface/sector, seek, rotation
  - **Hide complexity** of hardware devices
- Facilitate efficient use of storage device
  - Optimise access, e.g., to disk
- Support sharing
  - File persist even when owner/creator is not currently active (unlike the main memory)
  - Key issue: Provide protection (control access)

Hierarchical View of File System



Directory management:
- map logical name to unique Id, file descriptor

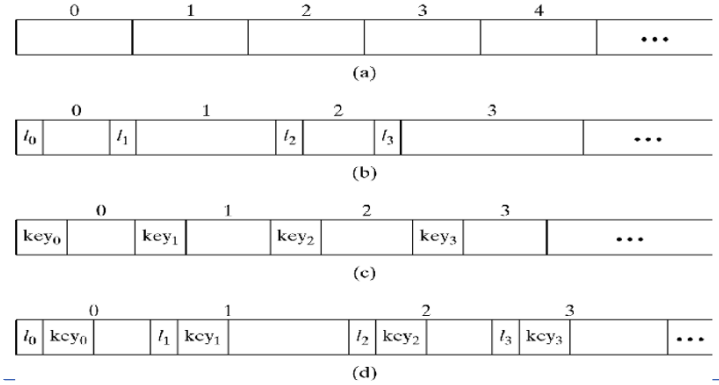
Basic file system:
- open/close files

Physical device organization:
- **map file data to disk blocks**

17.2 Files and Directories

User's View of File

- **File name and type**
  - **Valid** name
    - Number of characters
  - Extension
    - Tied to type of file
    - Used by applications
  - File **type** recorded in header
    - Cannot be changed (even when extension changes)
    - Basic types: **text, object, load file, dictionary**
    - Application-specific types: .doc, .ps, .html
- **Logical file organisation**
  - Most common: **Byte Stream**
    - 在文件系统中，**字节流（Byte Stream）** 指的是文件数据的一种表示方式，它将文件内容表示为一个连续的、顺序的字节序列。使用字节流的概念，文件系统不需要关心文件数据的具体含义或结构，而是把文件当作一个由单个字节组成的长串数据。
      字节流模型提供了一种简单抽象，使得读写文件操作可以不受文件数据类型的影响，因为所有文件都是以相同的方式处理—即一系列的字节。这种抽象同时适用于文本文件、二进制文件或任何其他类型的文件。
  - Fixed-size or variable-size **records**
    文件可以按记录（record）组织，每个记录存储一组相关数据。记录可以是固定大小的，也可以是可变大小的：
    - **固定大小记录**：每个记录占用相同数量的字节，这简化了记录的查找和访问，因为每个记录的起始位置可以轻易计算出来。
    - **可变大小记录**：记录根据存储的数据量有不同的大小。这种方式更灵活，但访问特定记录可能需要额外的信息或遍历整个文件。
  - Addressed
    - Implicitly (sequential access to next record)
    - Explicitly by position (record number) or key
    - **隐式地址化（Sequential Access）**：文件可以按照顺序访问，每次访问下一个记录。这种方式简单，适用于需要顺序处理文件的场景。
    - **显式地址化**：可以通过以下方式显式访问记录：
      - **按位置（Record Number）**：可以直接根据记录的编号访问，这通常适用于固定大小记录。
      - **按键（Key）**：特别是在记录有特定键（如数据库中的主键）的情况下，可以通过键来查找和访问记录。



Operations on Files

- Create/Delete
- Open/Close
- Read/Write (Sequential or Direct)
- Seek/Rewind (sequential) 寻找/倒退
- Copy (Physical or Link)
- Rename
- Change protection
- Get properties
- Each involves parts of Directory Management, BFS, or Device Organisation
- GUI is built on top of these functions

Directory Management

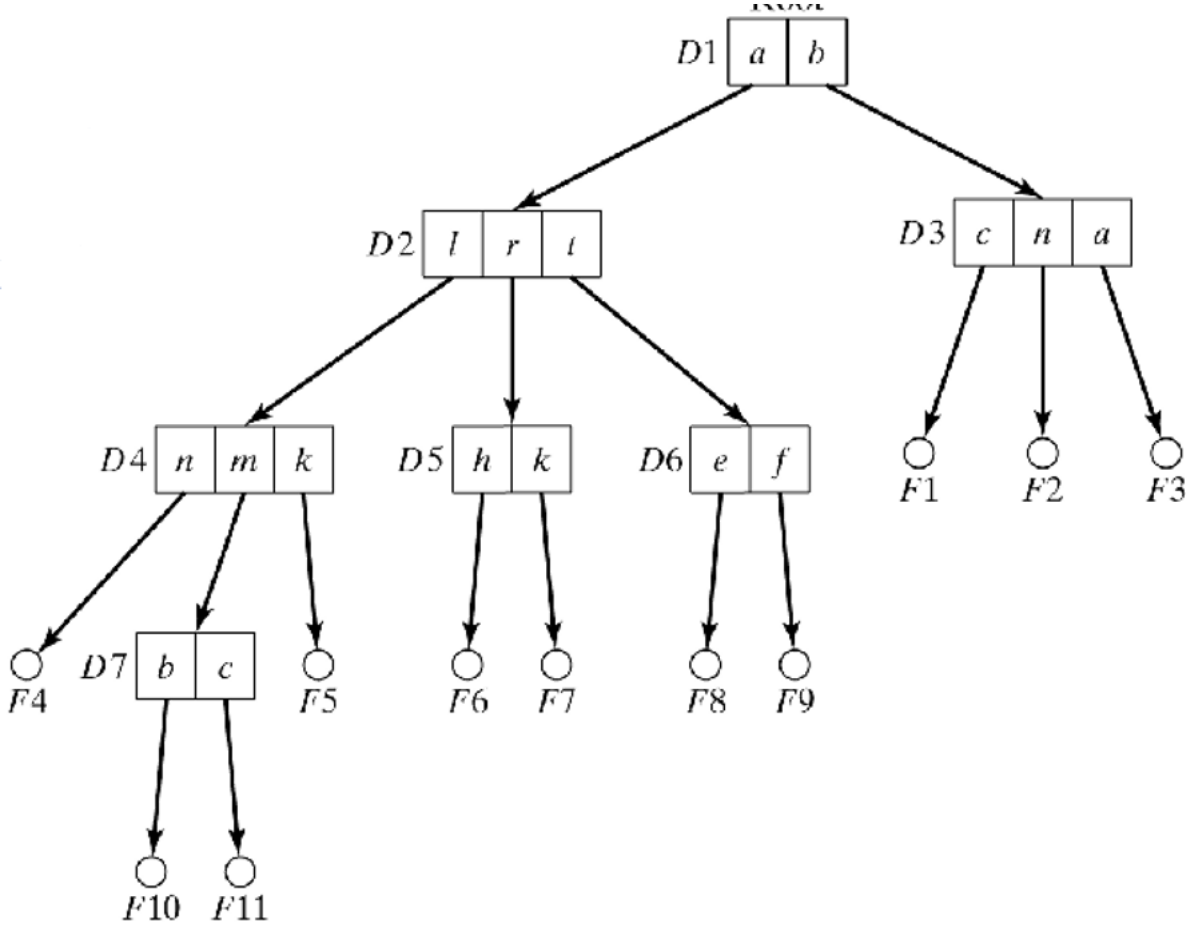
- Main issues:
  - **Shape** of data (e.g., tree, shortcuts)
  - **What** info to keep about files?
  - **Where** to keep it?
  - **How** to organise entries for efficiency?
- 1. **数据的结构（Shape）**：
  - 文件系统的数据结构可以采取多种形式，如树形结构、图形结构等。这影响了文件和目录的组织方式，以及用户如何导航和管理文件。

2. **关于文件的信息（What）**：
   - 决定存储哪些元数据来描述文件，例如文件大小、创建和修改日期、权限、所有者信息等。
3. **信息存储位置（Where）**：
   - 确定这些信息存储在文件系统的哪个部分。通常，这些信息存储在文件的元数据区域，可以是文件头部、专门的目录节点（inode）或其他数据结构。
4. **如何组织条目以提高效率（How）**：
   - 如何在目录中组织文件和目录条目，以便可以快速地插入、删除和搜索文件。

File Directories

- **Tree-structured**
  - Simple search, insert, delete operations
  - Sharing is asymmetric (only one parent)
    树形结构（Tree-structured）：
- 树形结构是一种常见的文件系统结构，它模拟了现实世界的文件夹和文件组织方式。
- 文件系统的每一项（可以是文件或目录）都位于一个树状的层次结构中，每个项有一个父目录（除了根目录），并且可以有多个子项。
  树形结构的特点包括：

- **简单的搜索、插入和删除操作**：
  - 树结构可以通过路径名来定位任何项，这使得搜索（查找文件）、插入（添加新文件）和删除（移除文件）操作相对简单。
- **非对称共享**：
  - 在纯粹的树形结构中，每个文件或目录只有一个父节点。这意味着共享（如创建快捷方式或链接）是非对称的，文件通常只能在一个地方拥有“真实”位置。

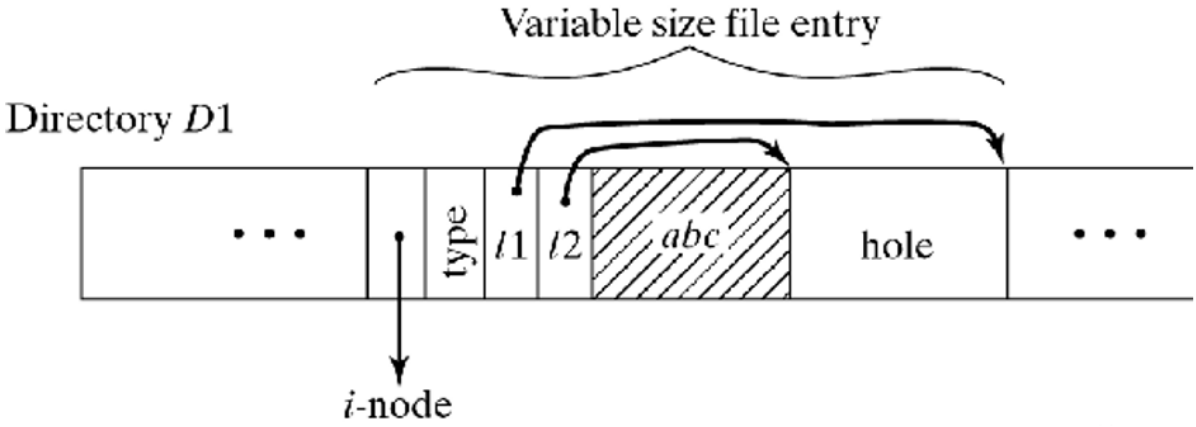


File Directories

- How to uniquely name a file in the hierarchy?
- **Path names**
  - Concatenated local names with delimiter
    `.` or `/` or `\`
  - **Absolute** path name: start with root
    `/`
  - **Relative** path name: start with current directory
    `.`
  - Notation to move upward in hierarchy (parent folder)
    `..`

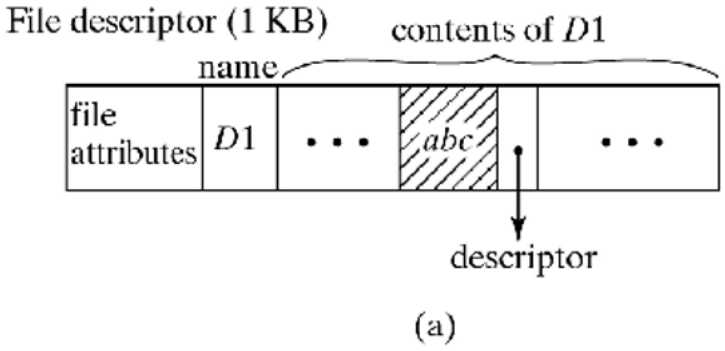
Implementation of Directories

- What information to keep in each entry
  - **All** descriptive information
    - Directory can become very large
    - Searches are difficult/slow
  - Only symbolic **name (filename) and pointer** to descriptor (file's properties)
    - Needs an extra disk access to descriptor

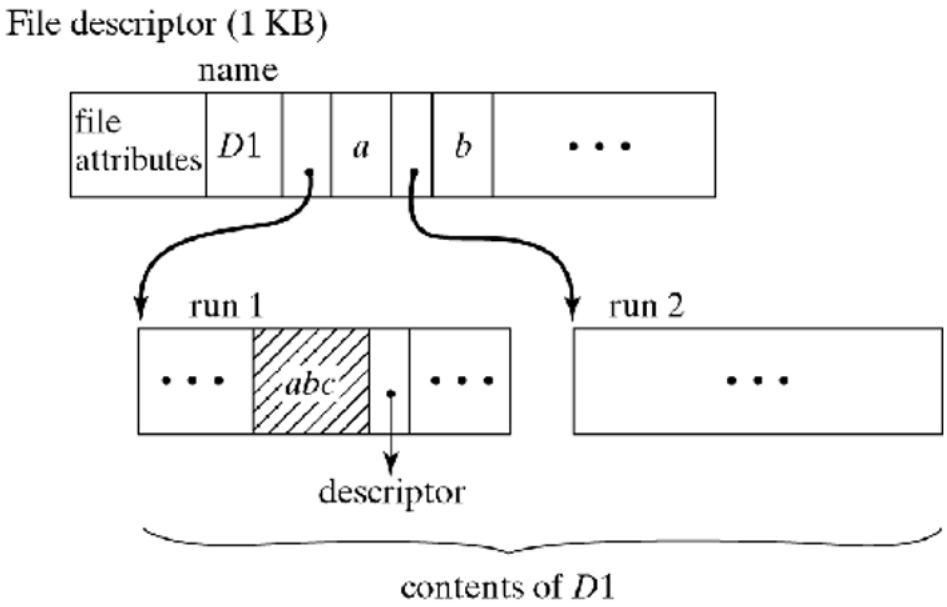


- How to organise entries within directory
  - Fixed-size entries
    use array of slots
  - Variable-size entries
    use linked list
  - Size of directory:
    fixed or expanding
- 目录可以有固定的大小，也可以设计为可扩展的：
  - **固定大小**的目录简化了管理，但可能限制了目录能包含的最大文件数。

- **可扩展大小**的目录更加灵活，可以随着文件的添加而增长，但可能需要更复杂的管理策略以处理目录大小的变化。



(a)



(b)

File Operations (re-visited)

- Assume that:
  - descriptors are in a dedicated area
  - directory entries have name and pointer only
- `create`
  - find free descriptor, enter attributes
  - find free slot in directory, enter name/pointer
- `rename`
  - search directory, change the name
- `delete`
  - search directory, free entry, descriptor, and data blocks
- `copy`
  - similar as create, then find and copy contents of file
- change protection

- search directory, change entry

17.3 Basic File System

- Open/Close files
  - Retrieve and set up information for **efficient** access:
    - get file descriptor
    - manage open file table
      打开/关闭文件：

1. **打开文件**：
   - 打开文件的过程包括检索文件的相关信息，并为文件访问设置必要的数据结构。这些信息通常存储在文件描述符或索引节点（i-node）中。
   - 为了有效地访问文件，操作系统会在打开文件时获取文件描述符，并可能在内存中管理一个打开文件表，以跟踪哪些文件当前是打开状态。
2. **关闭文件**：
   - 关闭文件意味着完成对文件的访问，并更新系统中的相关数据结构，如关闭文件表中的项，并可能更新文件的最后访问时间。
   - 关闭文件操作确保所有的数据都正确写入存储，并释放相关的系统资源。

- File descriptor (i-node in UNIX)
  - Owner id
  - File type
  - Protection information
  - Mapping to physical disk blocks
  - Time of creation, last use, last modification
  - Reference counter
- Open File Table (OFT) keeps track of currently open files
- open command:
  1. Search directory for given file
  2. Verify access right
  3. Allocate OFT entry
  4. Allocate read/write buffers
  5. Fill in OFT entry
     - Initialisation (e.g., current position)
     - Information from descriptor (e.g., file length, disk location)
     - Pointers to allocated buffers
  6. Return OFT index
- close command:
  1. Flush modified buffers to disk
  2. Release buffers

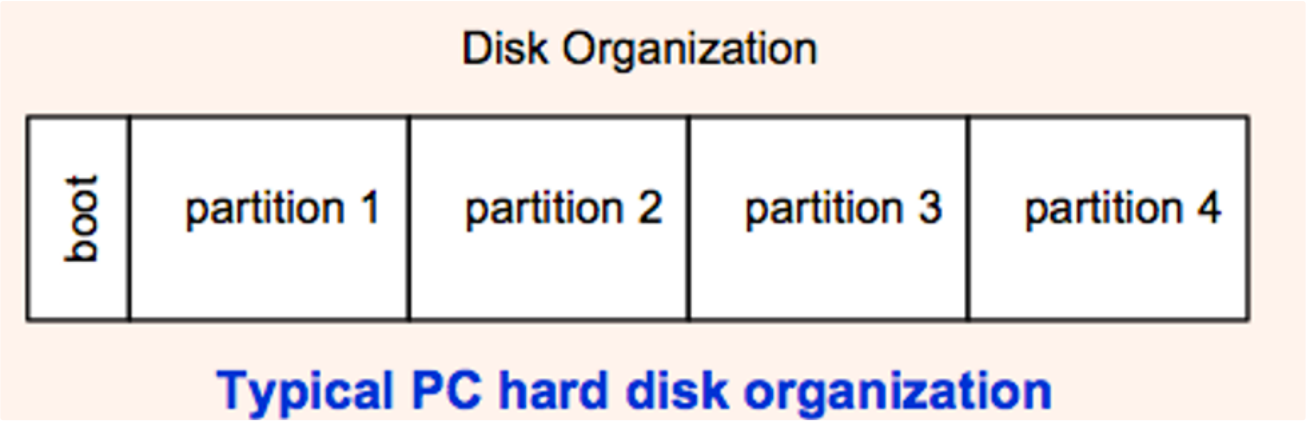
3. Update file descriptor
4. Free OFT entry

打开文件表（Open File Table）是操作系统用来管理所有当前打开文件的数据结构。每当一个进程打开一个文件时，操作系统会在这个全局表中创建一个条目，以跟踪文件的重要信息和状态。打开文件表是操作系统用来实现文件访问控制和文件共享的机制之一。
打开文件表通常包含以下信息：

- **文件描述符（File Descriptor）** 或 **文件句柄（File Handle）**：这是一个唯一标识符，关联到打开文件表中特定的条目。
- **文件位置指针**：它记录了文件内的当前位置，用于读写操作的下一次开始位置。
- **文件打开模式**：例如，只读、只写或读写模式。
- **文件的访问权限**：这些权限决定了哪些操作是允许的。
- **文件锁定状态**：指示文件或其部分是否被锁定，以及锁定的类型。
- **引用计数**：记录有多少进程正在使用这个文件。每当有新的进程打开同一文件时，引用计数会增加；当进程关闭文件时，计数减少。
- **指向文件控制块或i-node的指针**：这包含了文件所有的元数据，如文件大小、所有者、权限以及文件数据在磁盘上的位置等。

当程序执行打开文件操作时，操作系统首先检查文件的权限，然后在打开文件表中创建或更新一个条目，并将文件描述符返回给程序。程序随后使用这个文件描述符来进行读写等操作。
当文件关闭时，操作系统会更新打开文件表，并减少引用计数。当引用计数降至零时，操作系统会清理该条目，释放资源。

Organising Data on a Disk



- **Boot**: usually start of disk which contains code to **start loading** the OS (booting the OS). For PC, boot block also called **MBR** (Master Boot Record) and contains also the partition table
- Partitions: divide up disk into regions for filesystems, each filesystem in one partition. For PCs, only 4 **primary partitions** are allowed which can be booted
  - PC can also have **logical partitions** beyond primary partitions

Organising Files

- 在计算机存储中，扇区（Sector）和逻辑块（Logical Block）是数据存储和访问的两个基本单位，它们之间有着密切的关系：
  - **扇区 Sector:**
    - 扇区是硬盘驱动器（HDD）或固态驱动器（SSD）上的基本物理存储单位。
    - 在传统的硬盘上，一个扇区通常存储512字节数据；现代硬盘通常使用4096字节（4KB）作为一个扇区的大小。
    - 扇区是磁盘读写操作的最小物理单位。
  - **逻辑块（Logical Block）或文件系统块（Filesystem Block）：**
    - 逻辑块是文件系统管理数据的逻辑单位。
    - Logical blocks need not to be same as physical sector size - may be some multiple of sectors
    - 文件系统将一组扇区逻辑地组合成一个块，块的大小通常是扇区大小的整数倍。
    - 在文件系统中，块是分配空间和管理文件的基本单位。

Data Structure on Disk

- Need data structure to record which block belongs to which part of file
  - E.g., data at positions 2020-4100 are in which blocks and which part of the block?
- How does data in file change?
  - Write some data at end of file
  - UNIX: write data into holes, but basically means can write anywhere
  - Decreases with truncation operation
- Data structure must also be stored on disk (persistent)
- Typical data structures: versions of list/ trees/ arrays

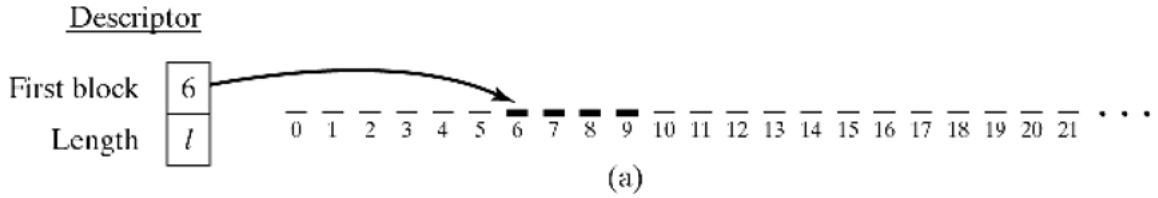
17.4 Physical Organisation Methods

Contiguous Organisation

连续组织（Contiguous Organization）是文件系统中文件存储的一种方法。在这种组织中，文件的所有数据块在磁盘上物理位置上是连续的。这意味着文件的起始块和结束块之间的所有块都是连续存放的，没有空隙。
连续组织的特点包括：

- **简单的访问**：因为文件的数据块是连续的，所以读取文件时可以一次性读取整个文件，这使得文件的访问速度非常快，特别是对于读取大文件时。
- **高效的磁盘利用**：连续的数据块减少了寻道时间，因为磁头不需要在不同的磁道之间移动来读取同一个文件的不同部分。
  连续组织的缺点：
- **碎片问题**：随着文件的创建和删除，磁盘上会出现碎片。新文件可能找不到足够大的连续空间，即使磁盘上总的空闲空间是足够的。

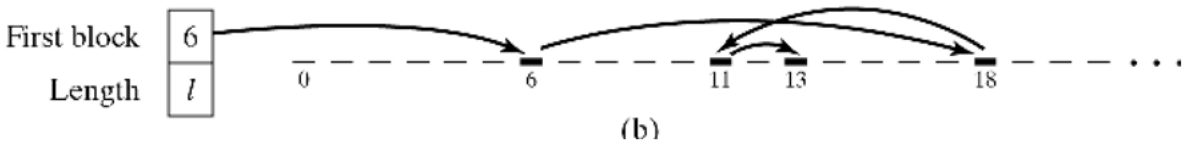
- **文件扩展问题**：如果一个文件需要扩展，但其后面的磁盘空间已被占用，该文件就需要移动到一个有足够连续空间的新位置。
- **预分配问题**：在创建文件时，必须预先知道文件的最大大小，这样才能分配连续的空间，这在许多情况下是不现实的。



Linked Organisation

链式组织（Linked Organization）是文件系统中文件存储的一种方法，其中文件的各个部分分散存储在磁盘上的不同区域，每部分通常被称为一个数据块或簇。在链式组织中，每个数据块都包含了指向文件下一个数据块的指针，形成了一个链条。这意味着，文件系统只需知道文件的起始块，就可以通过跟踪指针从一个数据块移动到下一个数据块，从而访问整个文件。
链式组织的特点：

- **非连续存储**：文件的数据块可以散布在磁盘的任何地方，不需要物理上连续。
- **动态扩展**：文件容易扩展，只需要在链的末尾添加新的数据块并更新前一个数据块中的指针。
- **无需预分配空间**：不需要在文件创建时就预知文件的最终大小，文件可以随着数据的添加而增长。
  链式组织的缺点：
- **访问速度较慢**：由于数据块可能分散在磁盘的各个部分，因此读取文件时可能需要多次寻道操作，这可能会导致访问速度较慢。
- **指针占用空间**：每个数据块都需要额外的空间来存储指向下一个数据块的指针。
- **可靠性问题**：如果链中的某个指针损坏，可能会导致整个文件的剩余部分无法访问。



Linked List Allocation - FAT

- MSDOS uses File Allocation Table (FAT)
- Linked allocation but stored completely in FAT (after reading from disk)
- FAT kept in **RAM** (stored in disk but duplicated in RAM) - gives fast access to the pointers
- FAT table contains either:
  - **block number** of next block
  - **EOF** code (corresponds to NULL pointer)
  - **FREE** code (block is unused)
  - **BAD** block (block is unusable, i.e. disk error)

    - Combines bitmap for free blocks with linked allocation for list of blocks
- FAT table is 1 entry for every block. Space management becomes an array method

FAT的基本概念：

- FAT是链式分配的一种形式，但与传统的链式分配不同，它不是将指针存储在数据块中，而是将整个链表存储在一个称为FAT的特殊表中。
- 在这个系统中，文件的每个数据块在FAT表中都有一个对应的条目。这个条目包含了文件的下一个数据块的编号或特殊代码。

FAT表存储在内存中：

- 虽然FAT表在磁盘上有物理副本，但为了快速访问，整个FAT表在系统启动时会被读取到RAM（随机访问存储器）中。
- 将FAT存储在RAM中可以显著提高访问文件数据块的速度。

FAT表的内容：

- **块编号（Block Number）**：如果文件占用了多个数据块，FAT表中的每个条目会指向文件的下一个数据块。
- **文件结束（EOF）代码**：表示文件结束的特殊代码，相当于链表中的NULL指针，标志着文件的最后一个数据块。
- **空闲（FREE）代码**：表明一个数据块当前未被使用。
- **坏块（BAD）标记**：表示数据块不可用，可能是因为磁盘错误。

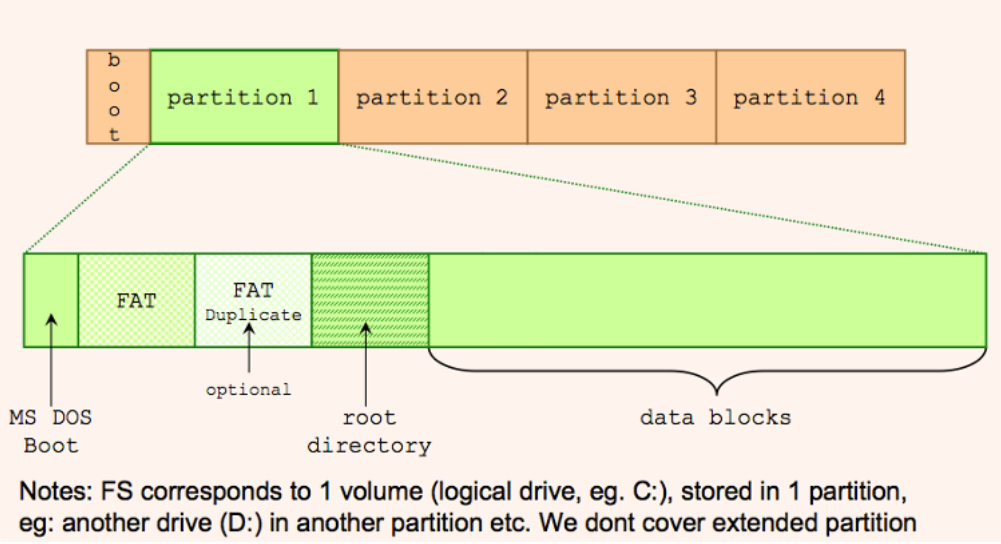
FAT的混合机制：

- FAT结合了位图（用于跟踪空闲数据块）和链式分配（用于管理文件的数据块列表）的特点。
- 它通过数组方式管理空间，每个数据块在FAT中都有一个对应的条目。

空间管理变成数组方法：

- 由于FAT表中为每个数据块都分配了一个条目，因此管理磁盘空间就像操作数组一样简单。可以通过索引FAT表来快速找到文件的连续块或检查空闲块。

MSDOS File System Layout



DOS Directory

- Special file (type directory) containing directory entries (32 byte structures, little endian)
- FAT directory entry: filename + extension (8 + 3 bytes), file **attributes** [read only, hidden, system, directory flag, archive, volume label, time+date created, last access date, time+date last write, first block (cluster) number]
- root directory is special (already known, FAT16 has limited root dir size, 512 entries), other directories distinguished by type

目录作为特殊文件：

- 在DOS中，目录被视为一种特殊类型的文件（类型为目录），它包含了一系列的目录项。
- 每个目录项是一个32字节的结构，使用小端字节序（Little Endian）格式存储信息。

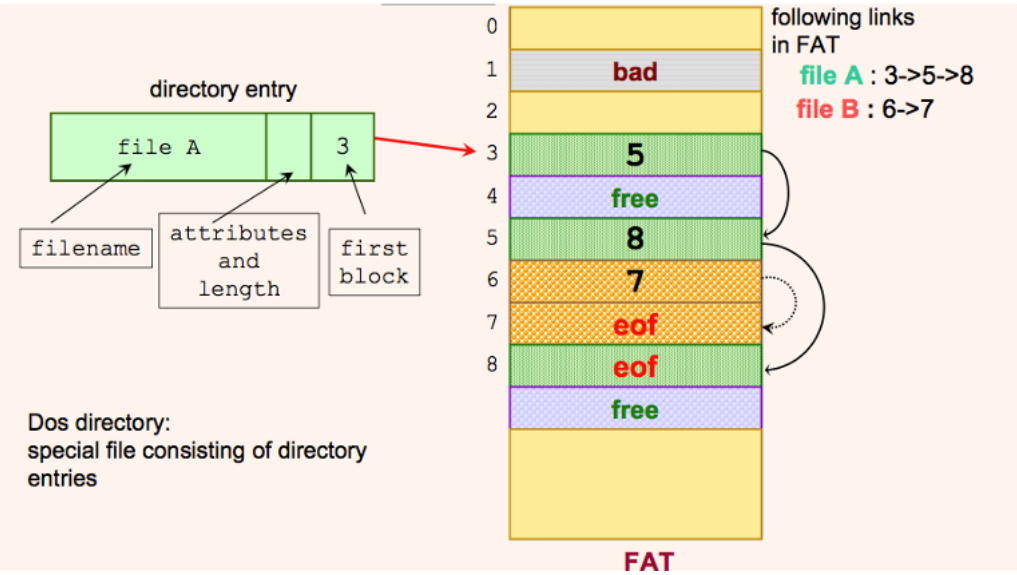
FAT目录项结构：

- **文件名和扩展名**：目录项的前11字节用于存储文件名（8字节）和扩展名（3字节）。
- **文件属性**：其中一部分字节用于存储文件属性，这些属性可能包括：
  - 只读（Read-Only）
  - 隐藏（Hidden）
  - 系统文件（System）
  - 目录标志（Directory Flag），用来标识该条目是一个文件还是一个目录
  - 存档（Archive），通常用于备份和文件变更跟踪
  - 卷标（Volume Label），标识磁盘卷的名称
- **时间和日期**：文件创建、上次访问以及上次修改的时间和日期信息。
- **第一个块（Cluster）号**：文件数据开始的地方，即第一个数据块或簇的编号。

根目录的特殊性：

- 根目录在FAT文件系统中有特殊的地位，因为它的位置是已知的（即在磁盘的固定位置）。
- 在FAT16文件系统中，根目录的大小是固定的，限制为512个条目。这意味着根目录可以直接存储512个文件或目录的信息，而不需要任何扩展。
- 其他目录则不受此限制，它们的大小可以动态变化，并且它们的条目可以通过目录类型的属性来区分。

FAT Organisation



- Linked list implemented as array of FAT entries:
  - 4 types:
    - Block number (part of a linked list)
    - EOF byte (end of linked list)
    - FREE byte
    - BAD byte
  - Some numbers are block numbers, some are reserved block numbers for EOF, FREE, BAD
- Blocks for file linked together in FAT entries, e.g., file A: 3→5→8
- Free blocks indicated by FREE entry
- Bad blocks (unusable blocks due to disk error) marked in FAT entry: BAD cluster value

FAT作为数组实现的链表：

- **FAT数组**：FAT表可以看作是一个大数组，每个数据块在数组中有一个对应的条目。文件系统通过这个数组来追踪文件存储的物理位置。

四种类型的FAT条目：

1. **块编号**：
   - 如果一个条目是块编号，那么它指向文件的下一个数据块。这些编号形成了一个链表，指示文件在磁盘上的存储顺序。
2. **EOF（文件结束）字节**：
   - 特殊的EOF标记表示文件数据块链表的结束。当FAT中的一个条目被标记为EOF时，它表示当前块是文件的最后一个数据块。
3. **FREE（空闲）字节**：
   - 如果一个条目被标记为FREE，那么相应的数据块当前没有被使用，可用于存储新数据。
4. **BAD（坏块）字节**：
   - 如果一个条目被标记为BAD，那么相应的数据块由于磁盘错误而不可用。

文件和块的链接：

- 文件的数据块在FAT中通过块编号链接起来。例如，文件A可能由FAT中的第3个块开始，第3个块的FAT条目指向第5个块，第5个块的条目又指向第8个块，形成链表3→5→8。

空闲块和坏块：

- 空闲块通过FAT条目中的FREE值来表示。文件系统在需要分配新块时会查找这些FREE标记的条目。
- 坏块通过FAT条目中的BAD值来标识。这些块不会被文件系统用来存储数据，因为它们可能会导致数据损坏。

MSDOS: Deleting a File

- How MSDOS delete file/directory:
  - Set first letter in filename to `0xE5` (destroys first byte of original filename)
  - Free data blocks: set FAT entries in linked list to FREE (tag all the data blocks in file free)

Delete 操作：
当在MS-DOS中执行删除操作时（例如使用 `del` 命令），文件系统通常执行以下步骤：

1. **标记FAT条目**：在文件分配表（FAT）中，文件的第一个簇（数据块）的条目被标记为“空闲”（FREE）。这意味着文件系统不再认为这些簇属于任何文件，因此它们可以被用来存储新数据。
2. **更新目录项**：文件的目录项中的文件名首字符通常被替换为一个特殊的删除字符（如0xE5），这样文件就不会在目录列表中显示出来。

Undelete 操作：
由于MS-DOS不会立即清除文件内容，而是只标记FAT中的簇为“空闲”并修改目录项，因此在新数据写入磁盘覆盖这些簇之前，文件通常可以被恢复。恢复文件的基本步骤包括：

1. **识别删除的文件**：通过扫描目录结构找到被标记为删除的文件项。
2. **检查FAT**：检查与该文件关联的簇在FAT中是否仍然是连续的，且没有被其他文件覆盖。如果这些簇未被覆盖，文件内容还在磁盘上。
3. **恢复目录项和FAT**：将目录项中的文件名首字符从删除字符恢复为原始字符，同时在FAT中恢复文件簇链的链接。

FAT16 Cluster Size

- FAT16: fat entry block number is **16 bits** (16 bit numbers in FAT entries), sector size = 512, how to deal with disks bigger than 64K * 512 (32M)
- Logical block size = multiple of sectors. MSDOS calls this the **cluster size**
- Maximum cluster size = 32K (normally)
- Maximum file system size of FAT16 = 64K * 32K = 2G
- Maximum file size: slightly less than 2G
- large cluster size means large internal fragmentation
- FAT16 can use 64K cluster size, now limits become 4G

FAT16 文件系统结构：

- 在FAT16文件系统中，每个FAT条目是16位的，这意味着FAT表中每个条目可以指向65,536（即2^16）个不同的簇。
- 扇区大小通常设置为512字节，这是硬盘存储的基本单位。

处理大于32MB磁盘的问题：

- 由于每个FAT条目是16位的，FAT16理论上可以直接管理的最大磁盘大小是32MB（即64K个簇乘以每个扇区512字节）。要管理更大的磁盘，需要增加每个簇包含的扇区数，即增加簇的大小。

簇大小：

- 簇是文件存储的逻辑块，它由多个扇区组成。MS-DOS将这个逻辑块称为簇。
- 簇的最大大小通常为32KB，这意味着每个簇可以包含64个扇区（因为每个扇区512字节）。

文件系统和文件大小限制：

- 如果每个簇为32KB，FAT16文件系统的最大容量为2GB（即64K个簇乘以每簇32KB）。
- 单个文件的最大大小略小于2GB，因为文件大小也受到簇大小和FAT条目数量的限制。

簇大小与内部碎片：

- 如果簇的大小较大，会导致较大的内部碎片，因为即使文件只比一个簇小一字节，也需要分配整个簇来存储它。

FAT16 的簇大小扩展：

- 尽管32KB是FAT16文件系统的常规最大簇大小，但在某些操作系统实现中，簇大小可以增加到64KB。这将文件系统的最大限制扩大到4GB，但会进一步增加内部碎片，并可能会导致与某些系统的兼容性问题。

VFAT

- VFAT: adds long filenames (255 chars)
  - Compatible with FAT16 by tricking it
  - short FAT16 filename for FAT16 and long version, short filename created from long one
  - For compatibility: long name stored as multiple directory entries using illegal attributes (not used by FAT16)
  - Trick: have to manage 2 kinds of names, old SW uses short names, alising issue due to 2 names

VFAT（Virtual File Allocation Table）是FAT文件系统（如FAT16）的扩展，它添加了对长文件名的支持。长文件名在Windows 95及以后的版本中得到了广泛使用。VFAT允许文件名最多达到255个字符，而FAT16只支持最多8个字符的文件名和最多3个字符的扩展名，即所谓的“8.3”命名约定。
如何实现长文件名支持：

- **与FAT16的兼容性**：VFAT通过在目录项中存储额外的信息来支持长文件名，同时保留了一个标准的FAT16短文件名条目。这样做确保了旧软件和操作系统仍然可以访问VFAT格式化的磁盘，即使它们不识别长文件名。
- **创建短文件名**：为了保持与FAT16的兼容性，VFAT自动为每个长文件名创建一个符合“8.3”命名约定的短文件名。这个短文件名是从长文件名派生的，可能包含一些特殊字符和数字来确保唯一性。
- **存储长文件名**：长文件名被分割成多个部分，并存储在一系列的目录项中，这些目录项在FAT16中使用非法属性标记，以防止FAT16系统将这些目录项当作常规文件或目录项处理。
- **多个目录项**：一个长文件名可能需要多个连续的目录项来完整存储。这些目录项中的每一个都包含文件名的一部分，并以特定方式链接起来，以便操作系统可以将它们重组为完整的文件名。

面临的挑战：

- **两种文件名的管理**：VFAT需要管理两种文件名，即短文件名和长文件名。这可能会在文件系统操作中引入复杂性。
- **别名问题**：因为每个文件有两个名字，可能导致别名问题，即两个不同的长文件名可能映射到相同的短文件名。
- **旧软件兼容性**：只认识短文件名的旧软件可能会遇到问题，因为它们可能无法正确处理长文件名。

FAT32

- Increase FAT size to 28 bits, cluster numbers 28-bits
- Max filesystem size increase: 127G
- **decrease internal fragmentation** (cluster size can decrease)
- **FAT table size increases**
- FAT16 root dir size limit removed - normal dir
- Maximum file size: $2^{32} - 1$

FAT32是FAT文件系统系列中的一个版本，它扩展了FAT表的大小，增加了文件系统能支持的最大容量，并对一些限制进行了改进。
FAT32的特性：

1. **FAT表大小**：
   - FAT32使用28位来索引簇（尽管总位数是32位，但实际只使用28位），这意味着FAT表可以跟踪更多的簇。这允许文件系统支持更大的存储设备。
2. **文件系统最大大小**：
   - 随着簇编号的扩展，FAT32文件系统理论上可以支持最大到127GB（甚至更大，取决于簇的大小）的存储设备。
3. **内部碎片减少**：
   - 由于可以支持更多的簇，FAT32允许使用更小的簇大小，从而减少了内部碎片—即未使用的存储空间，这通常发生在文件不够大以填满最后一个分配给它的簇。
4. **FAT表大小增加**：
   - 更多的簇意味着FAT表本身的大小会增加，因为需要更多的条目来映射磁盘上的簇。
5. **根目录大小限制**：
   - FAT32移除了FAT16中根目录大小的限制。在FAT32中，根目录被当作一个普通目录处理，可以动态增长，其大小受可用空间的限制。
6. **最大文件大小**：
   - FAT32支持的最大单个文件大小理论上可以达到2^32-1字节（即4GB减去1字节）。这对于大文件的存储提供了更大的灵活性。

Disk Fragmentation

磁盘碎片（Disk Fragmentation）是文件存储在硬盘上时出现的一种现象，它会影响硬盘访问速度和整体性能。这段内容描述了磁盘碎片的概念和产生的原因。

- Fast disk access:
  - Contiguous blocks (from geometry/processing viewpoint)
  - Blocks in same cylinder
- Suppose currently file system is optionally allocated (fresh install). Is this sustainable?
  - Delete files/blocks
  - Insert new files/blocks
- After some operations - block ordering becomes more random

- Disk Fragmentation: logical contiguous block are "far apart" on disk (this is different from memory fragmentation)

数据的快速访问：

- **连续块的快速访问**：如果一个文件的数据块在磁盘上是连续的，从几何和处理的角度来看，磁盘访问速度会更快。这是因为磁盘读写头不需要在不同的位置之间移动，可以一次性顺序读取所有数据块。
- **同一柱面上的块**：如果数据块位于硬盘的同一柱面上，那么磁盘的寻道时间会大大减少，从而提高数据访问速度。

碎片产生的过程：

- 在文件系统刚安装完毕时，数据可能会被选项地分配在磁盘上，这意味着文件会被存储在连续的数据块中。这时的文件系统（如新安装的FS）可能表现出优异的性能。
- 随着时间的推移，用户可能会删除一些文件和数据块，同时也会插入新的文件和数据块。这些操作会导致磁盘上的数据块分布变得更加随机和分散。
- 经过一系列的删除和写入操作后，文件的数据块可能不再连续，变得“分散”在磁盘上，这就是磁盘碎片。

磁盘碎片的影响：

- 碎片的产生意味着逻辑上连续的数据块在物理磁盘上实际上是“相隔甚远”的。这与内存碎片不同，内存碎片是由于分配和释放内存时产生的未使用空间的小块造成的。
- 磁盘碎片化导致硬盘读写头需要在磁盘上移动到多个不同的位置来读取一个文件，增加了寻道时间和旋转延迟，从而降低了数据访问速度。

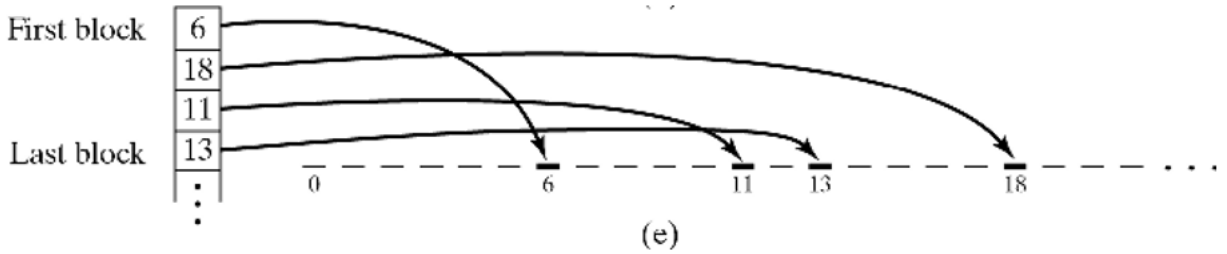
如何处理磁盘碎片：

- FAT:
  - Affects FAT Filesystem
  - fragmentation effect less with large cluster size (but **large** internal fragmentation)
  - MSDOS solution: run defragmentation (like compaction) on entire filesystem - move all used blocks to be contiguous
    - One big free space chunk after defragmentation.
    - May take a long time to defrag
- Unix S5FS:
  - also has disk fragmentation
  - may be worse than DOS since smaller logical block size

Indexed Organisation

- **Index table**: sequential list of records
- Simplest implementation: keep index list in descriptor
- Insert/Delete is easy
- Sequential and direct access is efficient
- **Drawback:** file size limited by number of index entries
  Variations of indexing
- Multi-level index hierarchy
  - Primary index points to secondary indices

- Problem: number of disk access increases with depth of hierarchy
- Incremental indexing
  - Fixed number of entries at top-level index
  - When insufficient, allocate additional index level



索引组织（Index Organization）是文件系统中用于管理文件存储的一种方法。在这种组织中，每个文件都有一个索引结构（如索引节点或inode），索引中包含了文件所有数据块的指针。这种方法允许文件的数据块在磁盘上非连续存放，同时仍然能够有效地访问文件的全部内容。

索引组织的特点：

1. **索引结构**：
   - 文件的索引结构通常存储在一个固定的位置，例如inode表中。
   - 索引中的每个条目对应文件的一个数据块或一组数据块。
   - 索引可以直接指向数据块，或者指向其他索引结构，这取决于文件的大小和文件系统的设计。
2. **直接和间接指针**：
   - 直接指针直接指向文件的数据块。
   - 大型文件可能还会使用间接指针，这些指针指向其他包含指针的块，这些指针再指向实际的数据块。
3. **支持大文件**：
   - 通过使用多级索引（如单级间接、双级间接、三级间接指针），索引组织可以支持非常大的文件。
4. **随机访问**：
   - 由于文件的每个数据块都可以通过索引快速定位，因此支持高效的随机访问。

索引组织的优势：

- **灵活性**：文件的数据块可以存储在磁盘的任意位置，不需要连续的空间。
- **效率**：对于小文件，通常只需要访问索引和少数几个数据块；对于大文件，虽然需要通过多个索引层次，但通常仍然比链式组织更高效。
- **可扩展性**：索引结构允许文件动态增长。

索引组织的缺点：

- **复杂性**：管理索引结构比链式组织或连续组织更复杂。
- **开销**：索引本身需要占用一定的磁盘空间，尤其是对于非常大的文件系统。

索引组织的变体涉及了不同的技术来管理文件在磁盘上的存储，尤其是大文件和数据库系统。这些变体的目的是在提高存储效率和减少访问延迟之间找到平衡。

多级索引层次结构（Multi-level Index Hierarchy）：

- 在多级索引结构中，一个主索引（Primary Index）不直接指向文件的数据块，而是指向次级索引（Secondary Indices），这些次级索引然后指向实际的数据块或进一步的索引层次。

- **优点**：这种方法允许文件系统管理非常大的文件，因为通过添加更多的索引层次可以极大地增加文件系统的最大文件大小。
- **缺点**：随着索引层次的增加，读取文件的某个部分可能需要更多的磁盘访问，因为每一级索引都可能涉及到一个磁盘I/O操作。

增量索引（Incremental Indexing）：

- 在增量索引方法中，顶级索引有固定数量的条目，这些条目直接指向文件的数据块。
- 当顶级索引的条目不足以包含所有数据块的指针时，文件系统会分配额外的索引层次来存储更多的指针。
- **优点**：这种方法在文件较小时保持了效率，只有当文件增长到需要额外索引层次时，才增加额外的开销。
- **缺点**：与多级索引层次结构类似，增量索引也可能导致随着文件的增长，磁盘访问的次数增加。

17.5 Unix File Systems

- Look at System V File System (s5fs) - simpler than modern FS implementations
- **inodes**: represents every file
- directories: contains names of files (recall maps filename to eventual file (hard link) or pathname (symbolic link))
- file allocation using a multi-level tree index
  System V文件系统的主要组成：

1. **索引节点（inodes）**：
   - 每个文件和目录在UNIX文件系统中都有一个对应的inode。inode包含了文件的元数据，如所有者、权限、大小、时间戳（创建、访问和修改时间）以及指向文件实际数据的指针。
   - inode编号是唯一的，它在文件系统中标识一个特定的文件。
2. **目录（directories）**：
   - 目录在UNIX文件系统中是特殊类型的文件，它们包含文件名和对应的inode编号。
   - 目录项将文件名映射到inode，这使得文件系统可以根据名称找到文件的inode，进而访问文件数据。
   - 目录可以包含硬链接（hard link）和符号链接（symbolic link）。硬链接直接关联到文件的inode，而符号链接则包含了指向另一个文件路径的文本指针。
3. **文件分配**：
   - UNIX文件系统使用多级索引树来分配文件数据。每个文件的inode包含直接指针（直接指向数据块的指针），以及一级、二级和三级间接指针（指向包含其他指针的块的指针）。这种结构允许文件系统有效地管理大文件和大量的小文件。

s5fs: inodes

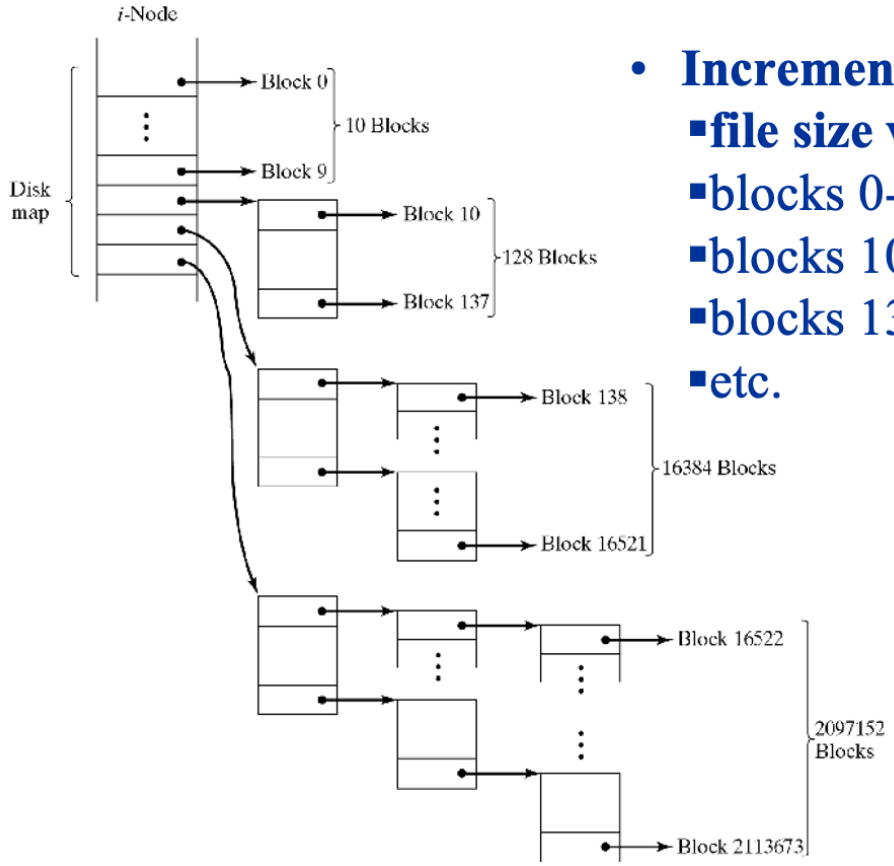
- actual file object
- every file has one inode (many to one mapping because of hard links)
- contains all metadata about file **except filename**
- contains reference count i.e. number of hard links, reference count = 0 means file can be deleted (subjected to no open files condition - all file descriptors to file object are closed)

- metadata in inode includes Table of Content (TOC) which gives mapping of file data to disk blocks (TOC is per file - contrast with MSDOS which has only **global** TOC (FAT))
  System V文件系统（s5fs）中的索引节点（inodes）是文件系统核心概念之一。它们代表了文件系统中的每个文件，并且承载了文件的大部分元数据。

inode的特性：

1. **文件的实际对象**：
   - inode代表了文件系统中的实际文件对象。每个文件在文件系统中都有一个唯一的inode。
2. **一对多映射**：
   - 由于硬链接的存在，多个文件名（位于不同的目录项中）可以映射到同一个inode。这意味着多个文件名可以引用同一个文件内容和属性。
3. **包含的元数据**：
   - inode包含了关于文件的所有元数据，除了文件名。这些元数据包括文件大小、所有者、权限、时间戳（文件创建、最后访问和修改时间）等。
   - inode还包含指向文件数据所在磁盘块的指针。这些指针构成了文件的“目录表”（Table of Content, TOC）。
4. **引用计数**：
   - inode包含一个引用计数，即指向该inode的硬链接数。当引用计数降至零时，意味着没有任何目录项引用这个inode，文件可以被删除。这当然也受限于文件是否被打开——即所有指向该文件对象的文件描述符都已关闭。
5. **每个文件的TOC**：
   - 与MS-DOS的FAT（全局文件分配表）不同，s5fs中的每个inode都有自己的TOC，这表示文件数据在磁盘上的位置。这提供了更灵活的文件数据管理方式，允许文件系统更有效地处理大文件和小文件。

Unix Table of Contents (TOC)



- **Incremental indexing in Unix**
  - **file size vs. speed of access**
  - **blocks 0-9: 1 access (direct)**
  - **blocks 10-137: 2 accesses**
  - **blocks 138-16521: 3 acc.**
  - **etc.**

s5fs: TOC Index Blocks

- **Direct block pointers:**
  used for small files, no extra disk overhead, efficient direct access. VM analogy: TLB
- **Single indirect block:**
  files bigger than direct blocks & smaller than double indirect. Disk overhead is 1 block. File access slightly slower than direct. VM analogy: direct mapped page table
- **Double + Triple indirect blocks:**
  files bigger than single indirect block. More disk overhead but is small fraction on file size. Random file access requires looking up the indirection blocks - slower than indirect. VM analog: 2-3 level page tables
- File sizes: direct << single indirect << double indirect << triple indirect

System V文件系统（s5fs）中的“目录表”（Table of Content, TOC）或者说索引块，是inode结构的一部分，它详细指出了文件的各个部分在磁盘上的位置。TOC使用了直接和间接的指针来定位文件的数据块。

TOC索引块的类型：

1. **直接块指针**：
   - 用于小文件的存储，因为直接块指针直接指向包含文件数据的磁盘块。
   - 对于直接块，没有额外的磁盘开销，文件访问效率高。
   - 虚拟内存（VM）的类比：这类似于转换后备缓冲区（Translation Lookaside Buffer, TLB）中的直接映射。
2. **单级间接块**：
   - 用于大于直接块容量但小于双级间接块容量的文件。

- 磁盘开销是一个块的大小，因为需要一个额外的块来存储指向实际数据块的指针。
- 文件访问速度比直接块慢，因为需要额外的步骤来解引用间接块。
- 虚拟内存的类比：相当于直接映射页表。

3. **双级和三级间接块**：
   - 用于大于单级间接块容量的文件。
   - 磁盘开销更大，因为需要多个间接块来存储额外级别的指针。
   - 随机文件访问需要查找多个间接块，这使得访问速度比单级间接块慢。
   - 文件的随机访问会更慢，因为需要遍历更多的间接指针。
   - 虚拟内存的类比：相当于2级或3级页表。

文件大小与索引块关系：

- 文件的大小决定了使用哪种类型的索引块：
  - **直接块**适用于最小的文件。
  - **单级间接块**用于中等大小的文件。
  - **双级间接块**和**三级间接块**用于更大的文件。

s5fs: File System Parameters

- **number of direct blocks** (e.g. s5fs: 10, ext2: 12)
- **number of indirection levels** (e.g. max is 2 or 3)
- **logical block size:** determines efficiency of disk I/O, can be composed of several contiguous physical blocks, space wastage from internal fragmentation, determines number of block pointers in index blocks and max indirection levels (e.g. ext2 can select 1,2,4 K when creating filesystem)
- **block pointer size:** affects indexing, determines max addressable disk block
- Small files only use the **direct blocks** in TOC , number of direct blocks can vary
- Larger file requires **first indirect block** (this is like 1-level page table)
- Even larger file uses **double indirect block** which points to indirect blocks. Even larger may have **triple indirect**
- Like page tables, can allow blocks which do not exist (**logical zero filled holes** in the file) - set the block pointer to NULL in the TOC, return zeroes for the logical block when read.

文件系统参数包括：

1. **直接块的数量**：
   - 文件系统为每个文件分配的直接块的数量，直接块用于存储文件的实际数据。s5fs可能使用10个直接块，而ext2使用12个直接块。
2. **间接层次的数量**：
   - 文件系统支持的最大间接索引层次，通常是2或3。每增加一个层次，都会增加可支持的文件最大大小。
3. **逻辑块大小**：
   - 确定磁盘I/O效率的一个关键参数。逻辑块可以由多个连续的物理块组成。逻辑块的大小还决定了索引块中可以包含的指针数量以及最大间接层次。

    - 例如，ext2文件系统在创建时可以选择1KB、2KB或4KB作为逻辑块的大小。

4. **块指针大小**：
    - 影响索引能力，确定了文件系统可寻址的最大磁盘块。块指针的大小必须足够容纳逻辑块的最大数量。

文件大小与索引结构的关系：

- **小文件**只使用TOC中的直接块，直接块的数量可以根据文件系统的设计而变化。
- **较大文件**需要使用第一个间接块，这类似于单级页表。
- **更大的文件**会使用双级间接块，双级间接块指向间接块，如果文件足够大，可能还会使用三级间接块。

文件系统设计的优化：

- 文件系统设计允许存在“逻辑零填充的空洞”—如果文件的一部分未被实际数据占用，相关的块指针可以设置为NULL。当读取这些逻辑块时，系统将返回零值，这避免了实际分配空间给未使用的数据。

s5fs: Directories

- A file of type directory - accessing directory is like any other file
- Only special directory operations allowed (read dir, link + unlink filenames)
- s5fs dir is array of 16 byte entries (inode: 16 bits = 2 bytes, filename 14 bytes)
- deleted file has inode 0

Inode Number	Filename
25	.
71	..
100	file1
200	file2

s5fs: Link + Unlink

- Create new file: new dir entry with new inode
- Hard link: new dir entry with inode of the linked file: e.g. `link(path1, path2)`
- Deleting (deleting is unlink since graph is DAG): remove dir entry, decrement inode link count, free file object when link count = 0 (plus open file condition)

s5fs: Conclusion

UNIX系统的System V文件系统（s5fs）是一个传统的文件系统模型，它为UNIX操作系统提供了基本的文件存储和管理机制。s5fs的设计简单且稳定，它使用了一些核心概念，这些概念在今天的许多文件系统中仍然存在。下面是s5fs的主要特点和组件的总结：

inodes

- s5fs使用inode（索引节点）来表示文件系统中的每个文件。
- 每个inode包含了关于文件的所有元数据，除了文件名。
- inode存储了文件的权限、所有者信息、大小、时间戳、以及文件数据的位置。
- 文件在目录中的名称通过目录项链接到其inode。

目录结构

- 目录在s5fs中被视为特殊类型的文件，它们包含文件名和相应的inode编号。
- 目录项映射文件名到实际的文件（硬链接）或路径名（符号链接）。

文件分配

- s5fs利用多级树索引结构来分配文件数据。
- 这包括直接块指针、单级间接块指针、双级和可能的三级间接块指针。
- 文件的数据块可以直接通过inode访问，或通过一个或多个间接索引层次访问。

文件系统参数

- 文件系统的性能和能力受到其参数的影响，如直接块的数量、间接层次的数量、逻辑块大小和块指针大小。
- 不同大小的文件使用不同的索引结构，从直接块到多级间接块。

优势与局限

- s5fs提供了一种简单有效的方式来管理文件和目录，支持了UNIX系统的基本需求。
- 它通过引用计数和inode机制，有效地处理了文件的创建、删除和硬链接。
- 尽管s5fs在管理大型文件和大容量存储设备方面有局限性，它为后续更复杂的文件系统设计奠定了基础。