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
```

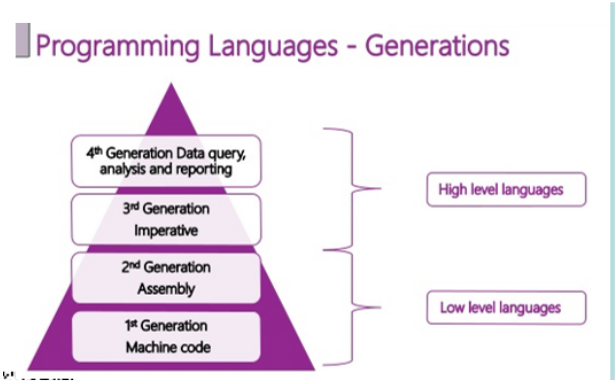
Machine code

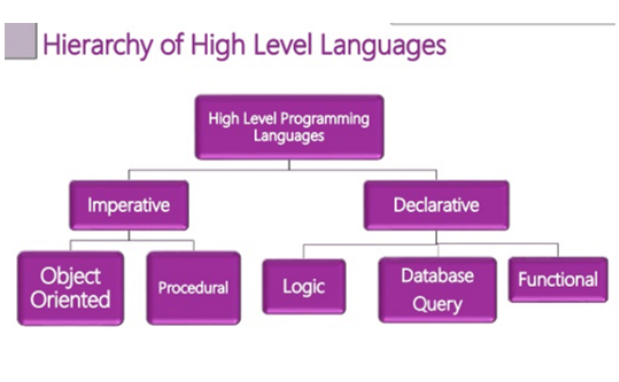
```
00100000000010010000000000001010
00000001001010010100100000100000
. . .
```

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

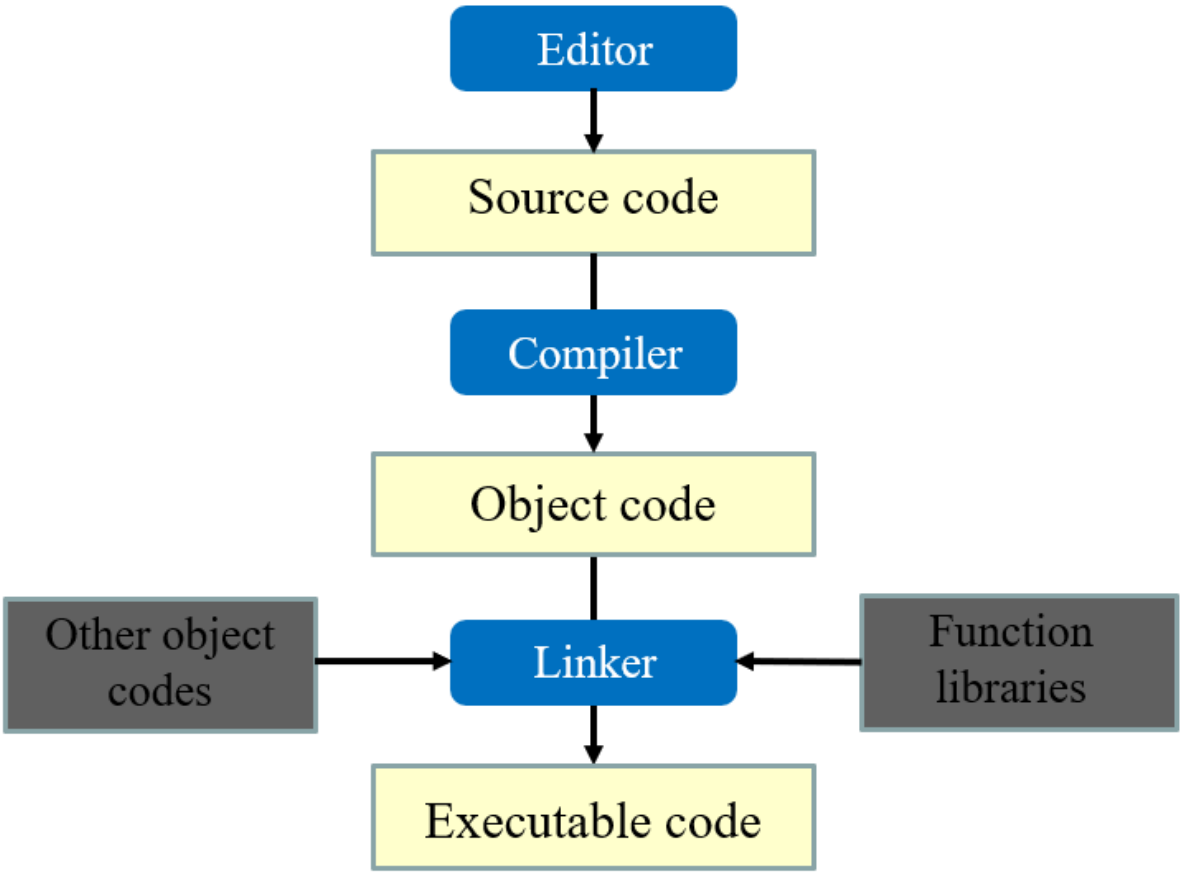
    - Declarative languages
    - Functional languages (Lisp, Scheme, SML)
    - Logic programming (Prolog)
- "Generational" classification of high level languages (3GL and later) was never fully precise
- A different classification is based on paradigm





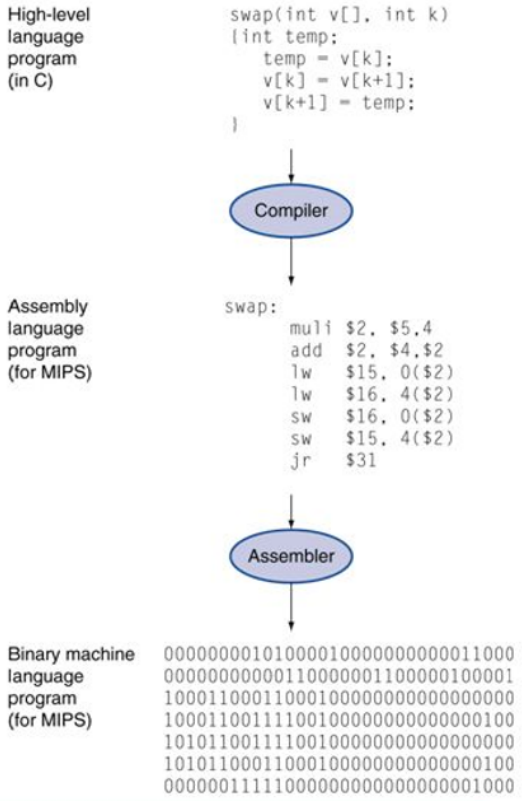
1.1.1 C Programming Language

- 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"

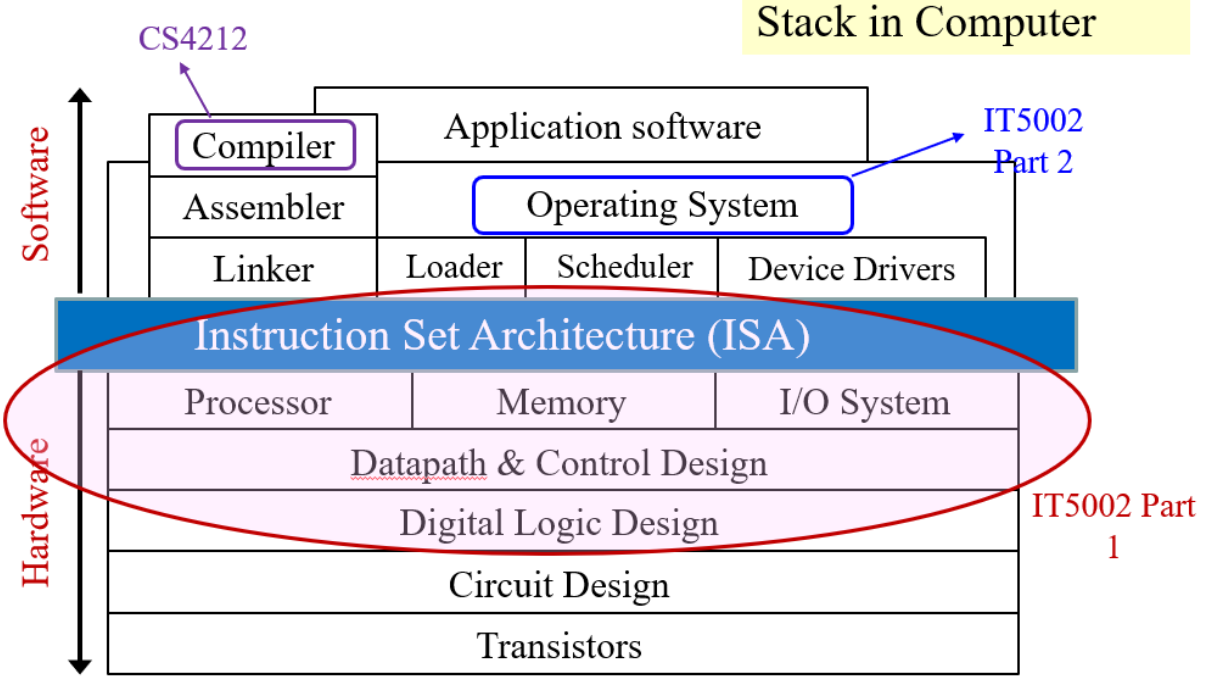


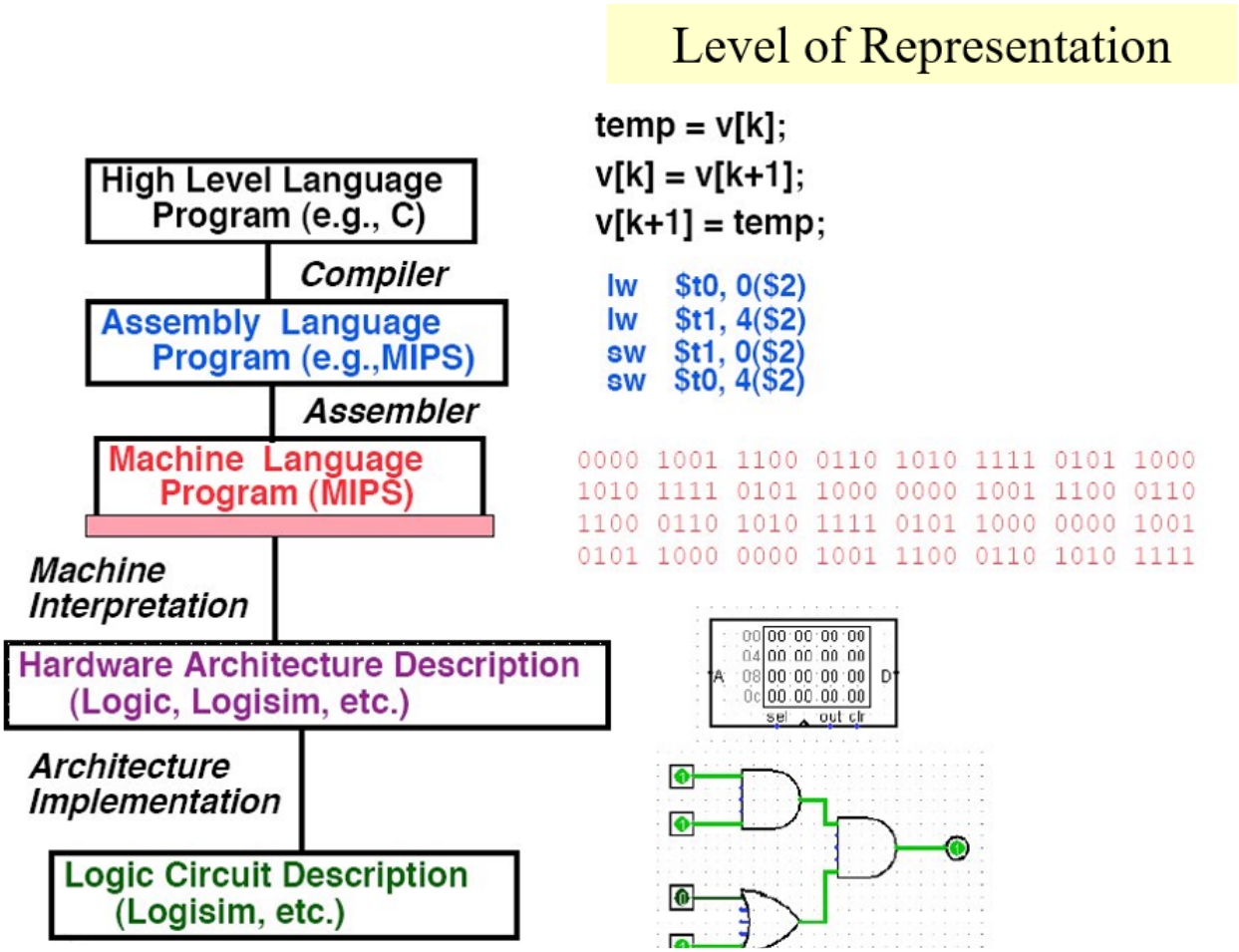
1.2 Abstraction

- 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

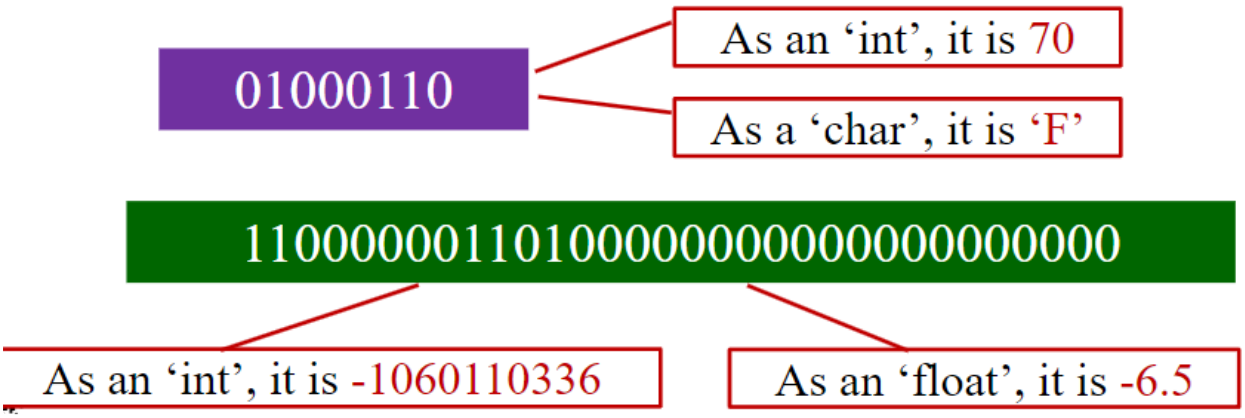




2 – Number Systems

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
  - $(7594.36)_{10} = (7 \times 10^3) + (5 \times 10^2) + (9 \times 10^1) + (4 \times 10^0) + (3 \times 10^{-1}) + (6 \times 10^{-2})$

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

- $1101.101_2 = 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-3} = 8 + 4 + 1 + 0.5 + 0.125 = 13.625_{10}$
- $572.6_8 = 5 \times 8^2 + 7 \times 8^1 + 2 \times 8^0 + 6 \times 8^{-1} = 320 + 56 + 2 + 0.75 = 378.75_8$
- $2A.8_{16} = 2 \times 16^1 + 10 \times 16^0 + 8 \times 16^{-1} = 32 + 10 + 0.5 = 42.5_{10}$

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

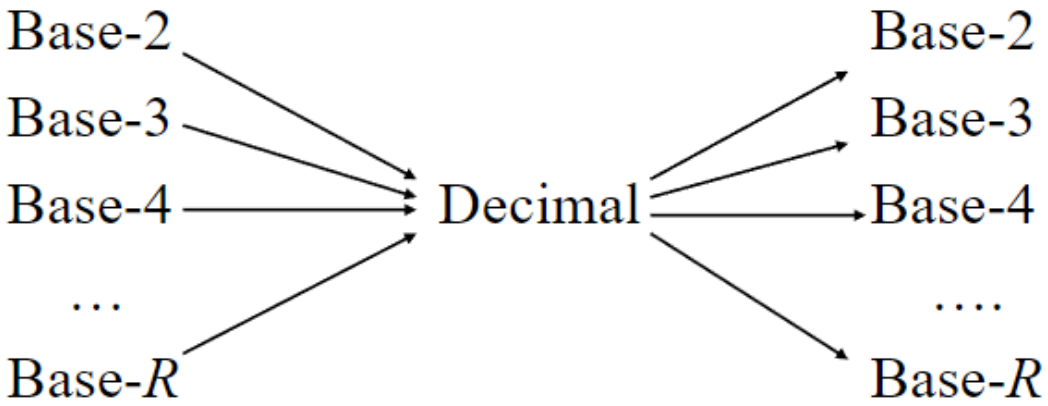
- **Base-R to decimal:** multiply digits with their corresponding weights
- **Decimal to binary (base 2)**
  - For whole numbers: repeated division-by-2 (2.5.1)
  - For fraction numbers: repeated multiplication-by-2 (2.5.2)
- **Decimal to base-R**
  - For whole numbers: repeated division-by-R
  - For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

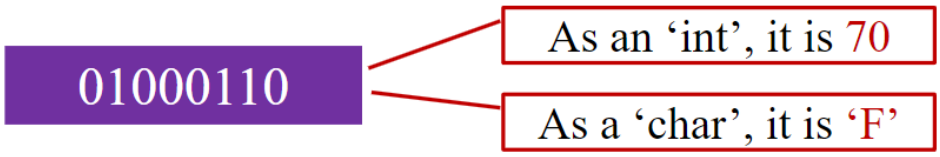
- **ASCII code** and **Unicode** are used to represent characters `('a', 'C', '?', '\0')`
- ASCII
  - American Standard Code for Information Interchange
  - 7 bits, plus 1 parity bit (odd or even parity)

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



CharAndInt.c

```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。

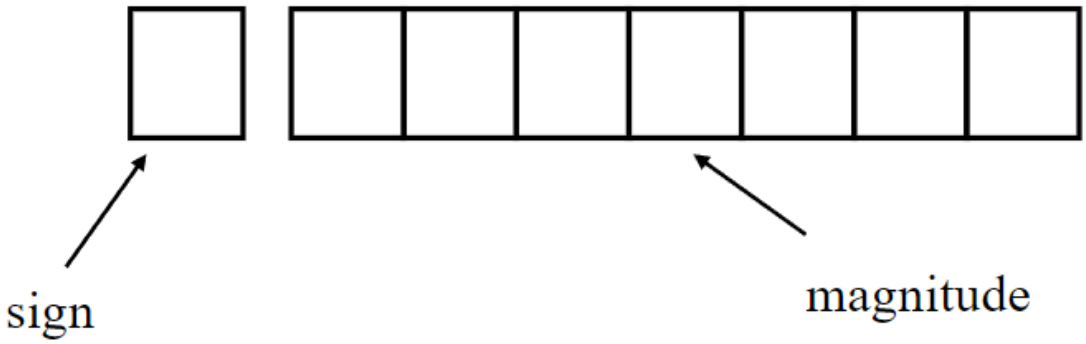
```
2,147,483,640 + 1 = 2,147,483,641
2,147,483,641 + 1 = 2,147,483,642
...
2,147,483,646 + 1 = 2,147,483,647  // 这是int的最大值
2,147,483,647 + 1 = -2,147,483,648 // 溢出，变成int的最小值
-2,147,483,648 + 1 = -2,147,483,647
-2,147,483,647 + 1 = -2,147,483,646
```

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` -> $+110100_2 = +52_{10}$
  - `10010011` -> $-10011_2 = -19_{10}$
- 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

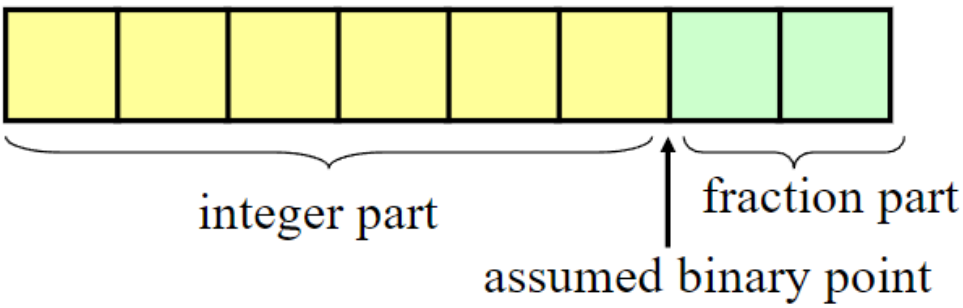
Excess-8 Representation	*Value*
1000	0
1001	1
1010	2
1011	3
1100	4
1101	5
1110	6
1111	7

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

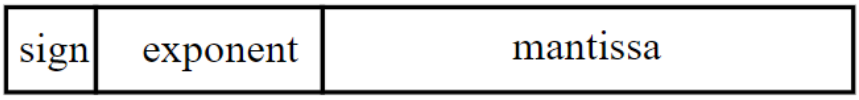


- If 2s-complement is used, we can represent values like:
  $011010.11_{2s} = 26.75_{10}$
  $111110.11_{2s} = -000001.01_2 = -1.25_{10}$

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)**

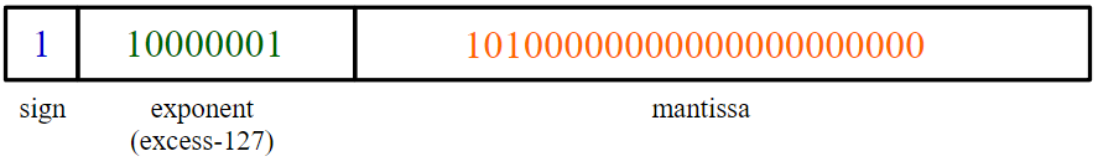


- Sign bit: 0 for positive, 1 for negative
- Mantissa is **normalized** with an implicit leading bit 1
  - 110.1_2 -> normalized -> $1.101_2 \times 2^2$ -> only 101 is stored in the mantissa field
  - 0.00101101_2 -> normalized -> $1.01101_2 \times 2^{-3}$ -> only 01101 is stored in the mantissa field

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

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

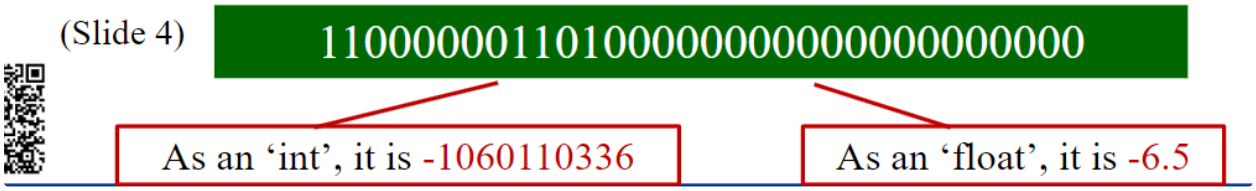
Exponent = 2 + 127 = 129 = 10000001_2



- 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。

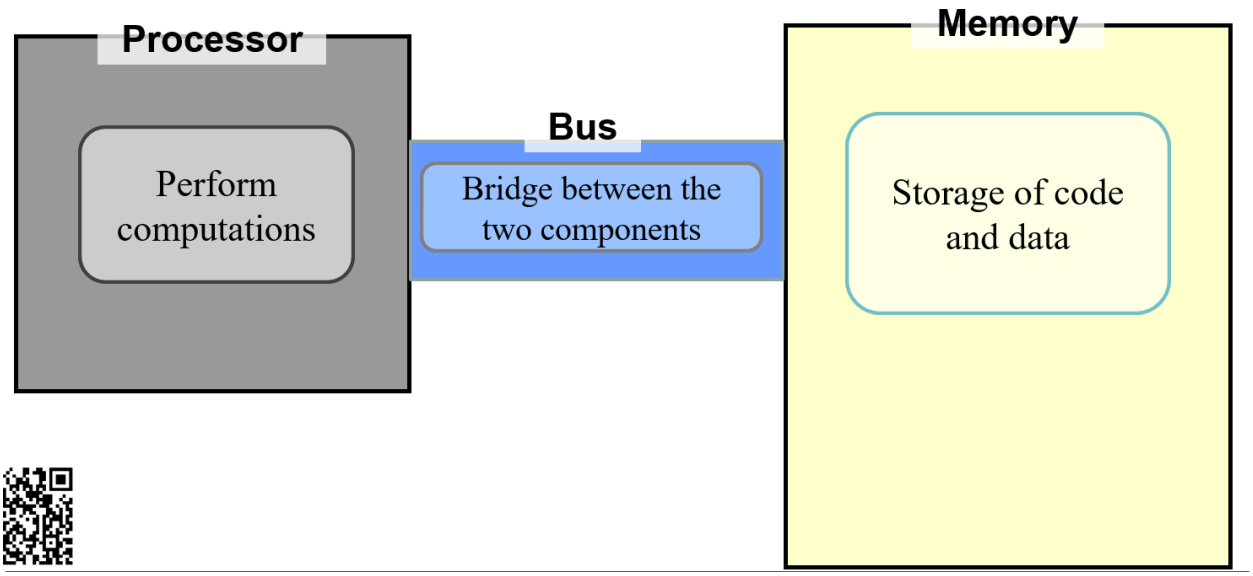
- Instruction Set Architecture (ISA) (指令集架构)
  - An abstraction on the interface between the hardware and the low-level software
    - Software: To be translated to the instruction set
    - Hardware: Implementing the instruction set
  - ISA Allows computer designers to talk about functions independently 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

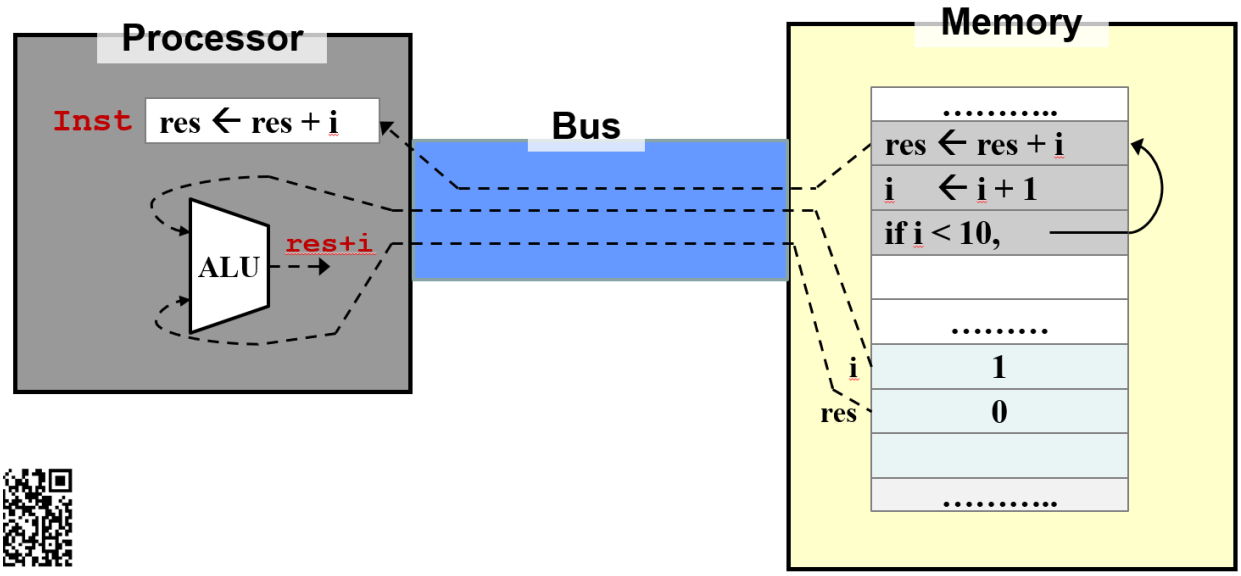


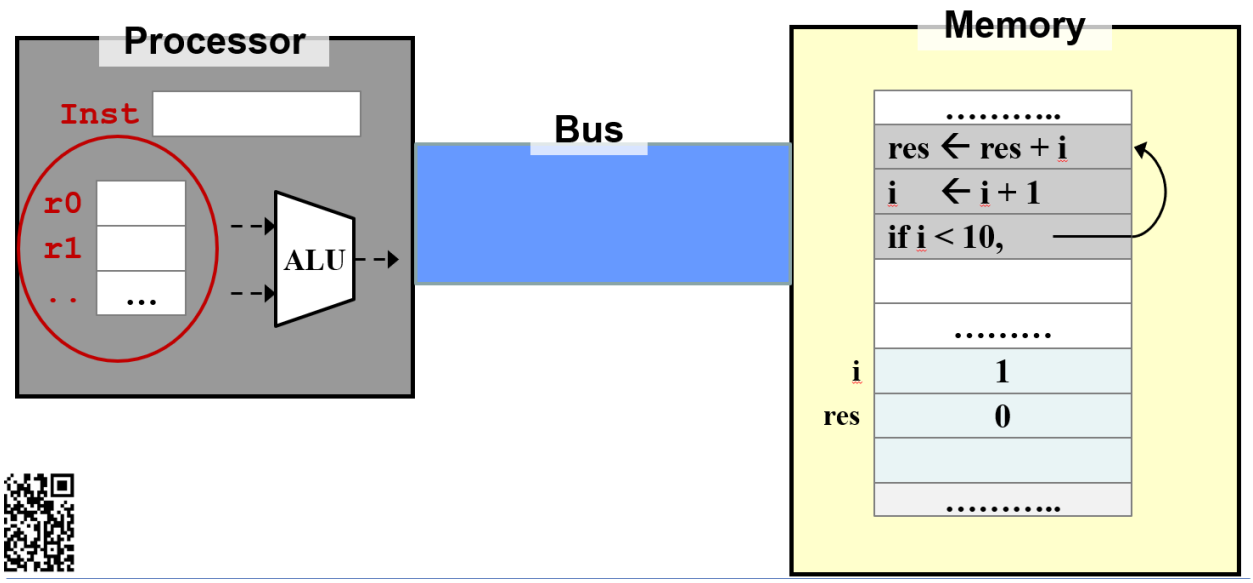
- Assume a simple computing-storage platform, including a processor, bus and a memory
  - Processor processing instructions
  - Bus transmit the data between memory and processor
  - Memory store the instructions and temp data

The code in Action

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

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

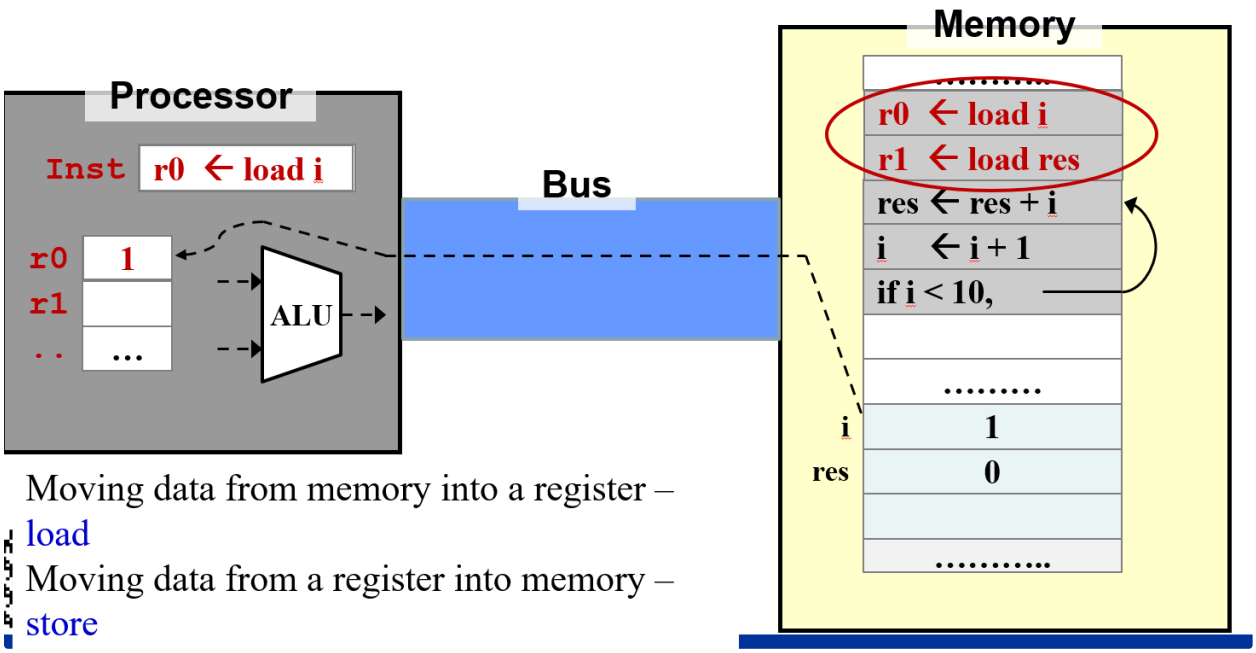




- The instructions and variables are stored in the **memory**
- The variable, `i` and `res` has been transmitted to **processor** through **bus**
  - The data of two variables and the instruction has been stored in **CPU register**, in order to get faster speed
  - > - The register (寄存器) is inside in 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

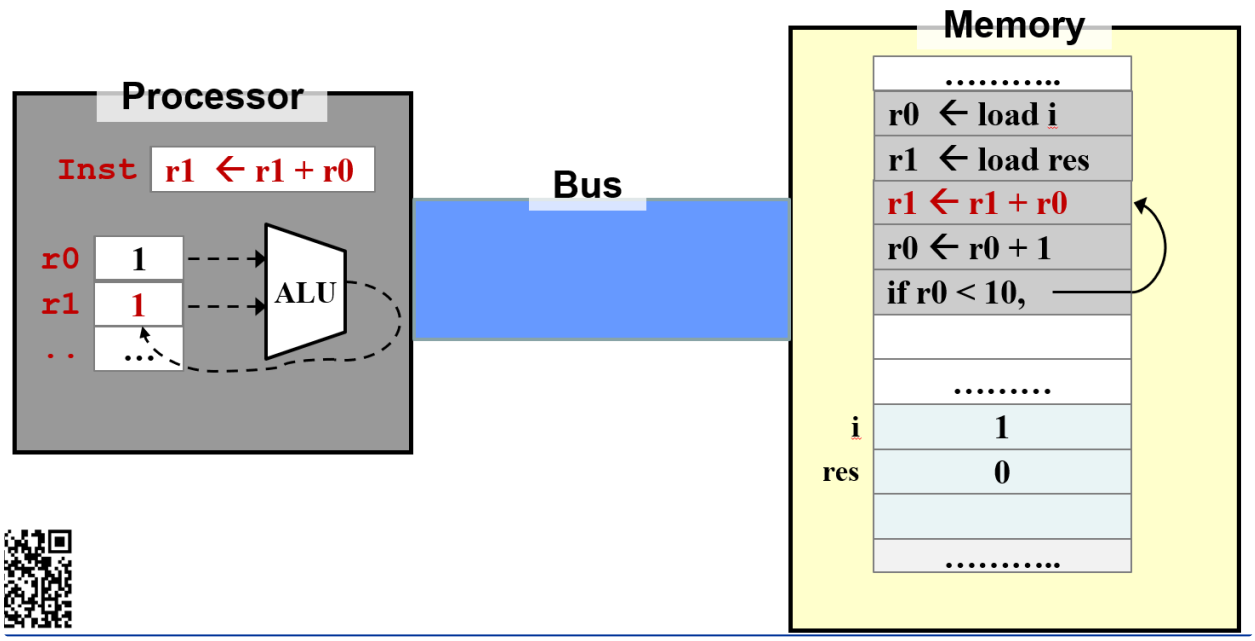


Moving data from memory into a register – load

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`

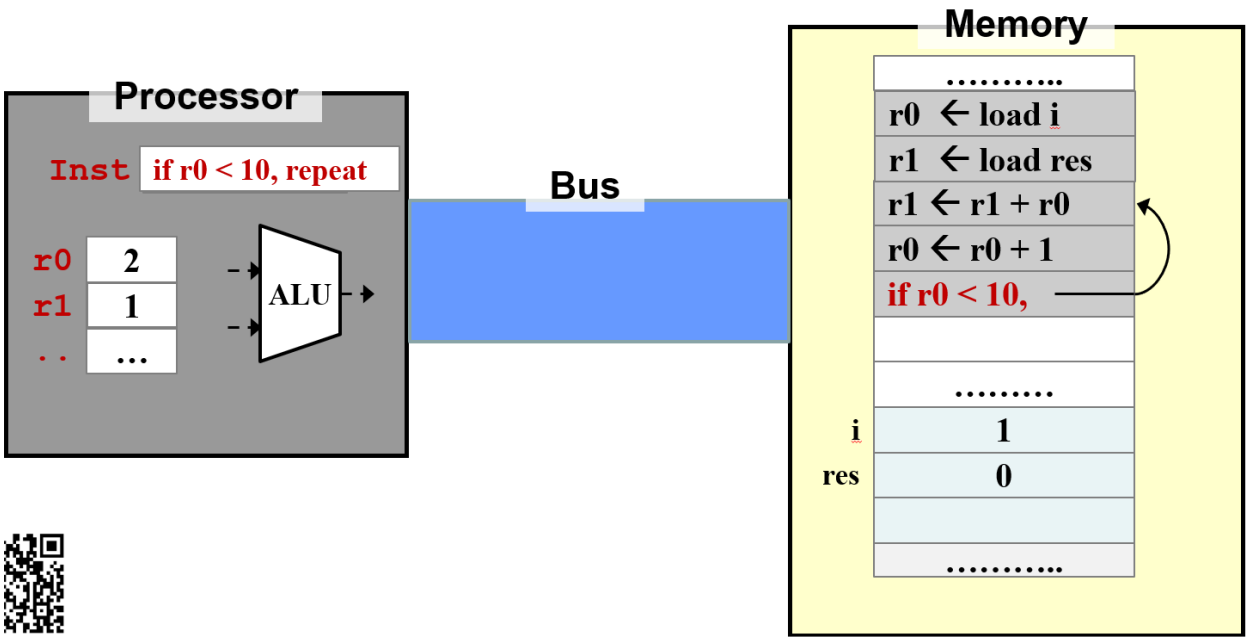
Reg-to-Reg Arithmetic



- 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

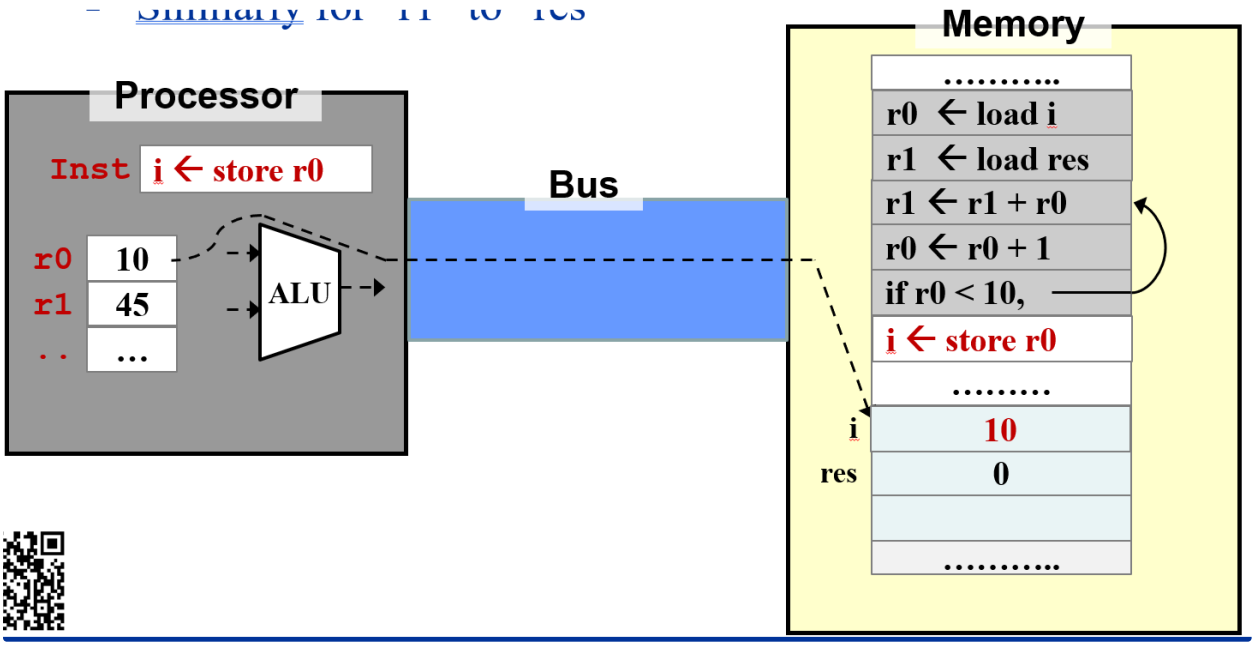
- Sometimes the arithmetic operation uses a constant value instead of register value
  - `r0 <- r0 + 1`, `1` is the constant value
  - > 常数被称为"立即数"（Immediate Value）。立即数是直接编码在机器指令中的常数值。
    >
    > 所以，当ALU要执行加法操作时，它不需要从寄存器或内存中加载立即数，因为这个值已经直接包含在指令中了。这意味着CPU可以在执行指令的同时，直接从指令本身获取这个常数值，并在ALU中与寄存器中的值进行计算。

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：

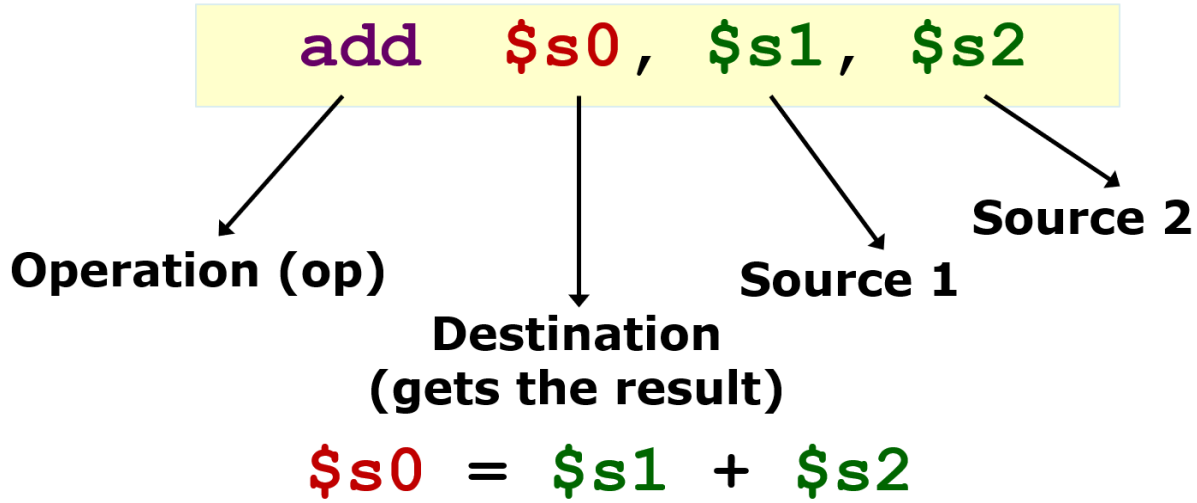
1. **指令集**：MIPS 指令集被设计得相对简单，并与其硬件管道设计紧密相连。这使得MIPS能高效地执行指令，同时简化了硬件实现。
2. **寄存器**：MIPS架构具有32个通用寄存器，标记为 `$0` 到 `$31` 。其中， `$0` 寄存器总是存储值0，其他寄存器用于不同的目的。例如， `$sp` 是堆栈指针， `$ra` 是返回地址寄存器。
3. **指令格式**：MIPS指令有几种格式，最常见的是R型、I型和J型。不同的格式支持不同的操作，例如算术运算、数据传输和跳转。
4. **指令示例**：以下是一些常见的MIPS汇编指令示例：
   - `add $t0, $t1, $t2` ：将 `$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

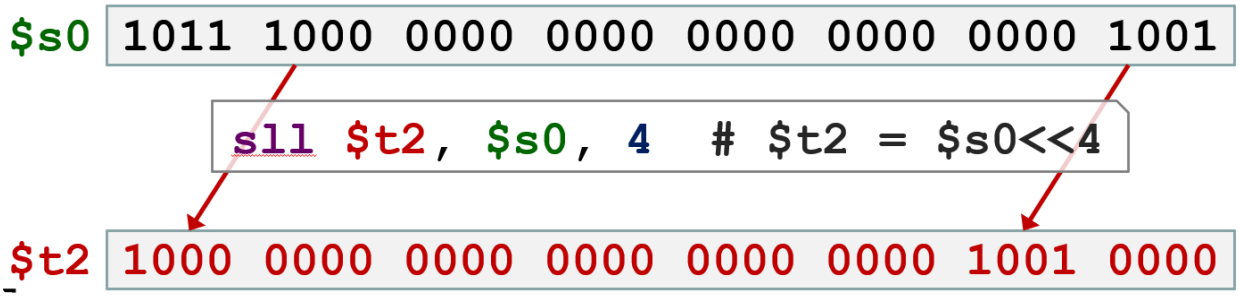
XOR

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`

	$t1	0110 0011 0010 1111 0000 1101 0101 1001
mask	$t2	0000 0000 0000 0000 0011 1100 0000 0000
	$t0	0000 0000 0000 0000 0000 1100 0000 0000

- 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

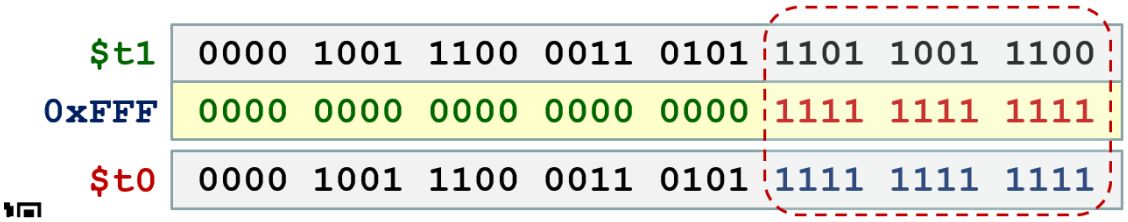
Bitwise OR

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

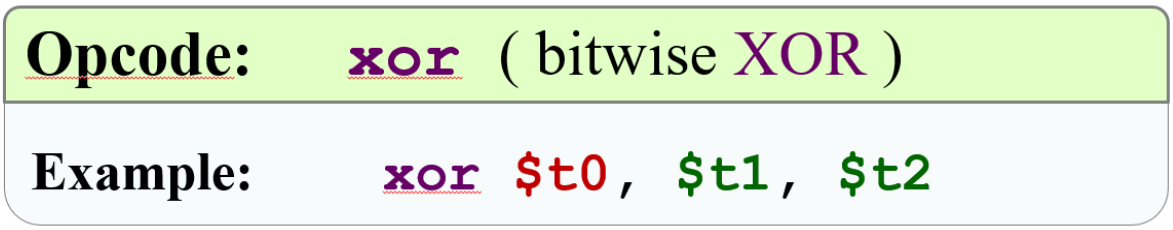
- Strange fact 1:
  - There is no **NOT** instruction in MIPS to toggle the bits (1 → 0, 0 → 1)
  - However, a **NOR** instruction is provided:

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



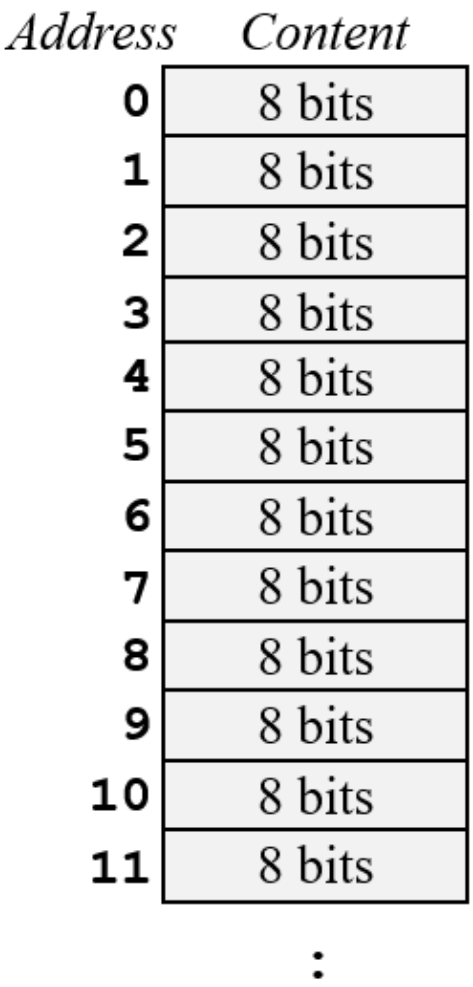
- Bitwise XOR: `xor`
- To get `not` operation through `xor`:
  - `xor \$t0, \$t0, \$t2`
- There DO exist `xori`

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

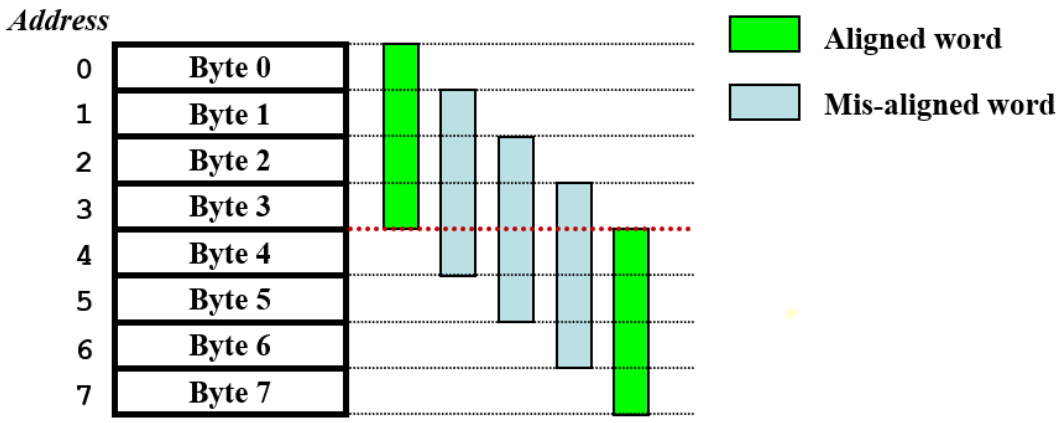
- Using distinct memory address, we can access:
  - a single byte (byte addressable) or
  - a single word (word addressable)
- Word is:

- 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)

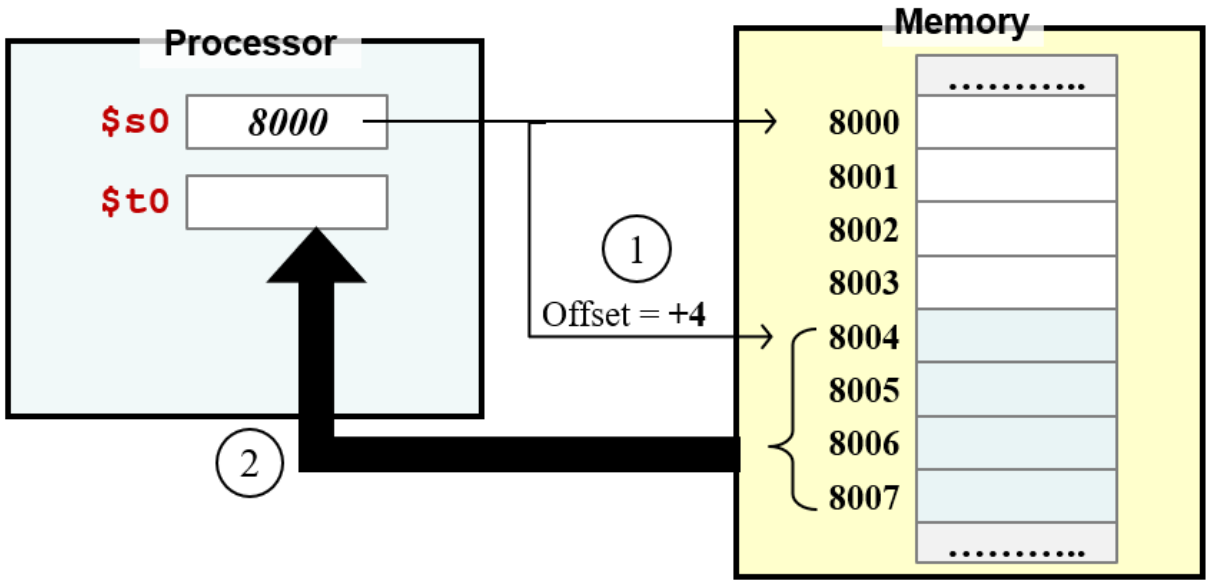
- Memory addresses are 32-bit long

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)`

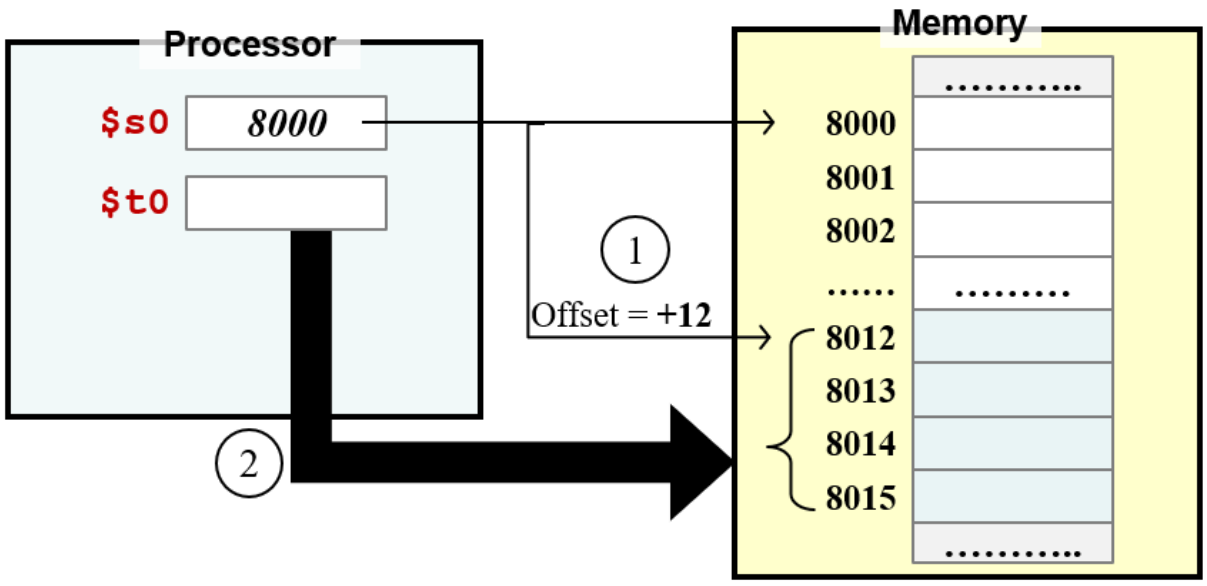


- Steps:
  1. Memory Address = `$s0 + 4 = 8000 + 4 = 8004`
  2. Memory word at `Mem[8004]` is loaded into `$t0`

> 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

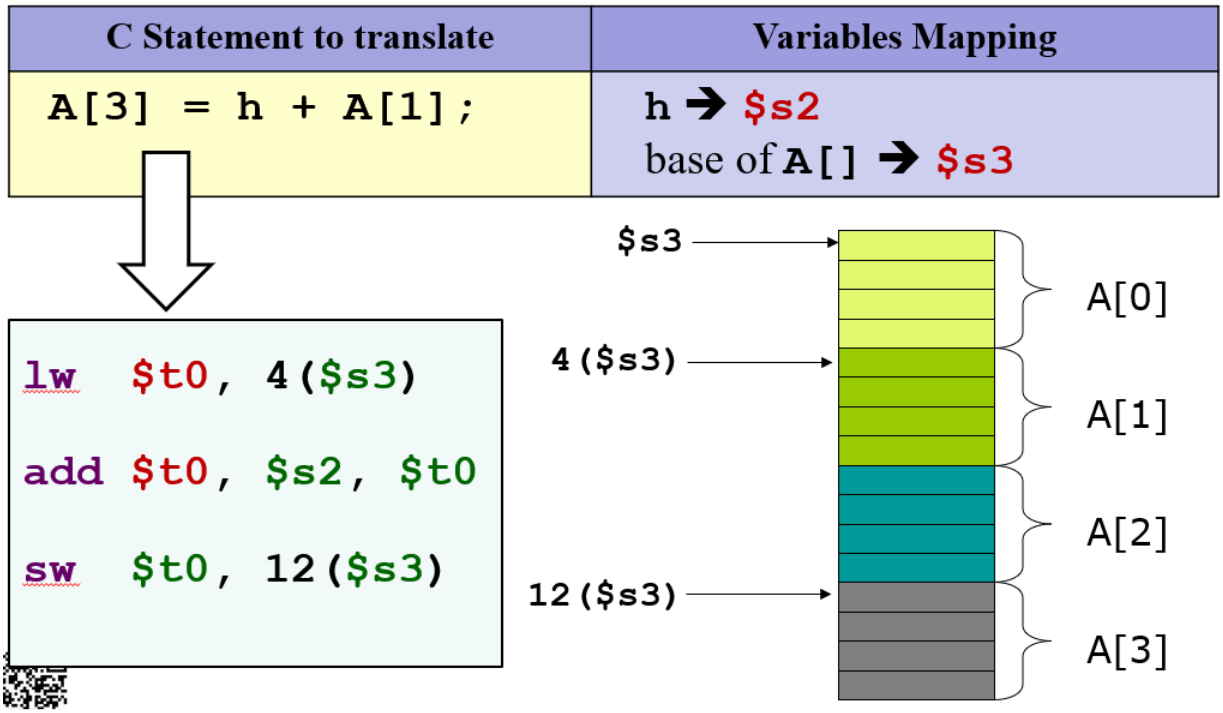
`A[7] = h + A[10]`	`lw $t0, 40($s3)`
	`add $t0, $s2, $t0`
	`sw $t0, 28($s3)`

- 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:
```

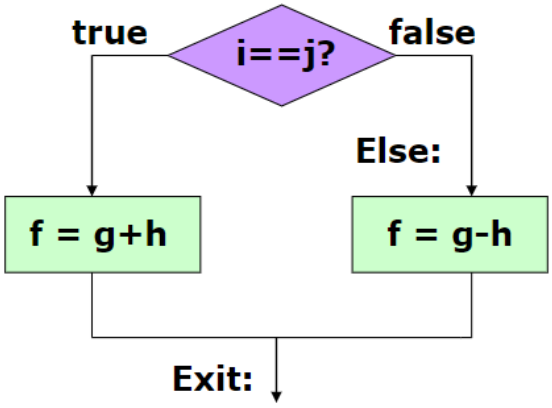
```
      bne $s3, $s4, Exit
      add $s0, $s1, $s2
Exit:
```

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

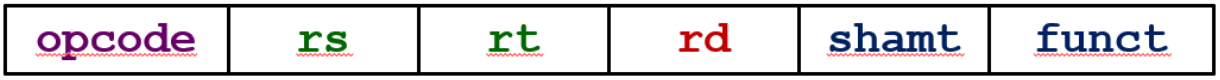
Name	Register number	Usage
$t8-$t9	24-25	More temporaries
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address

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:



- 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

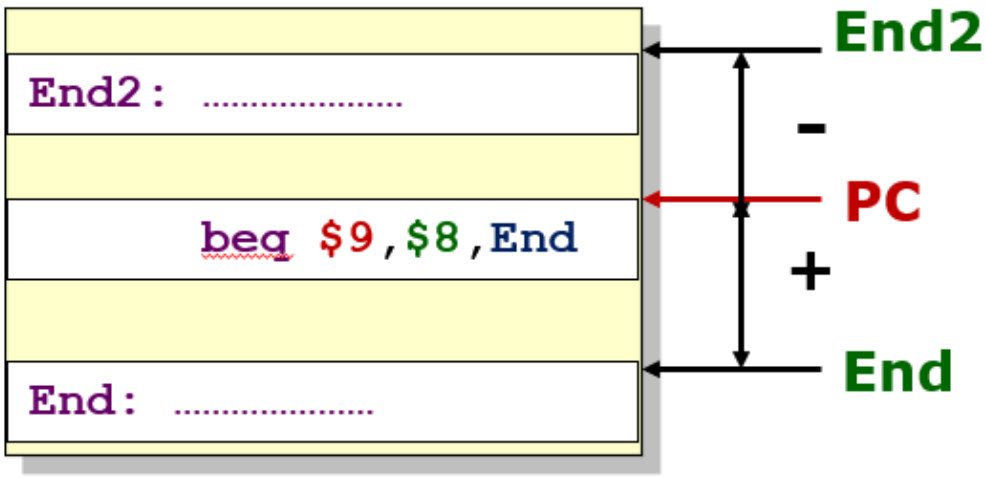
- As instructions are stored in memory, they too have addresses
  - Conrtol flow instructions uses these addresses
- As instructions are 32-bit long, instruction addresses are word-aligned as well
- Program Counter(PC) is a special register that keeps address of instruction being executed in the processor

> 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

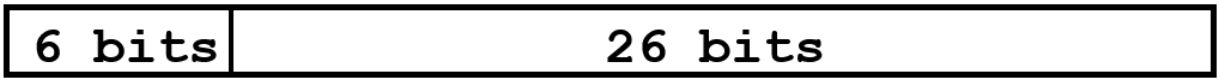
> 即如果没有进入branch，则跳转到下一个指令，由于每个MIPS指令是32 bits，所以在PC上加4字节
>
> 如果进入了branch，则跳转到一个新的地址来执行指令，这个新的地址是基于当前PC值，偏移量（即下一条指令的位置PC+4），以及由分支指令中的立即字段（ `immediate` ）给出的额外偏移量计算出来的
>
> 立即数字段（ `immediate` ）被解释为word的数量，而每个word有4字节，所以被乘以4

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

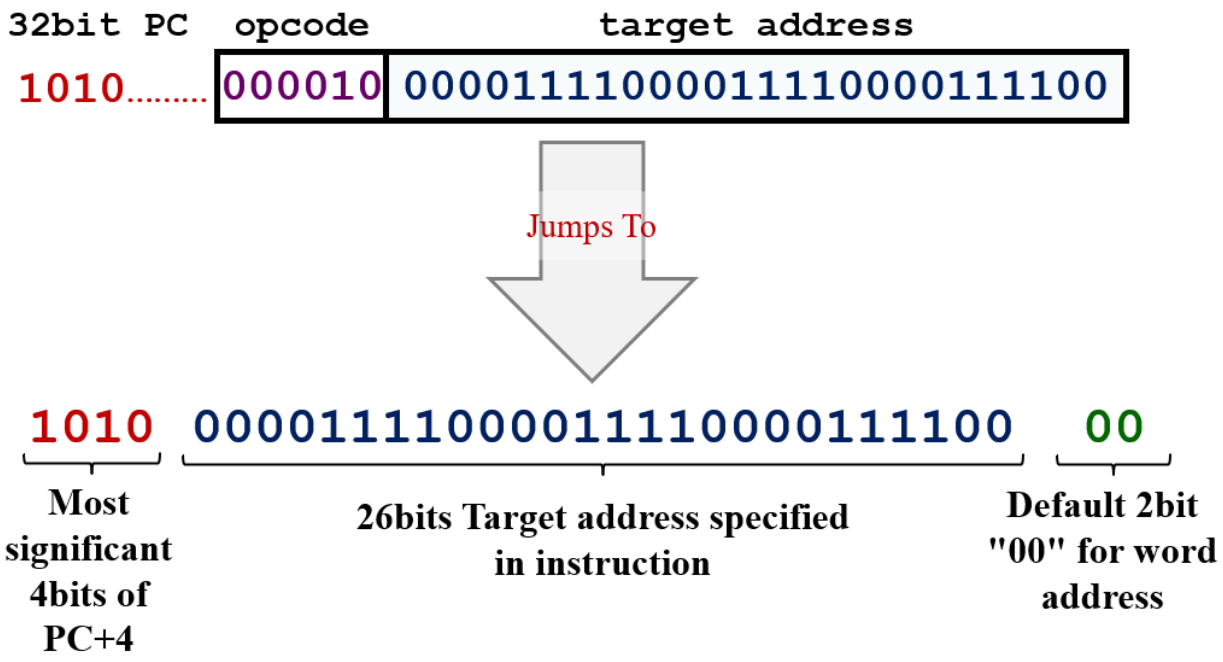


- Keep `opcode` field identical to R-format and I-format for consistency
- Combine all other fields to make room for larger target address
- We can only specify 26-bits of 32 bits address
- Optimization:
  - Just like with branches, jumps will only jump to word-aligned addresses, so last two bits are 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

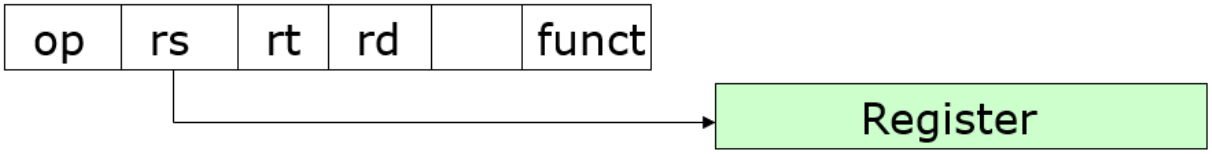
> 在MIPS中，使用J-format的 `j` 指令时，我们确实只有26位来指定跳转地址。这26位的地址是如何转化为完整的32位地址的呢？它是通过将这26位的跳转字段左移2位（因为每个指令是4字节或32位）来实现的，然后再与当前指令地址的高4位进行组合，从而生成完整的32位跳转地址。
>
> 因此，这26位地址可以指定 2^{26} 不同的跳转目标，也就是 `67,108,864` 个可能的目标。由于每个指令都是4字节，所以跳转范围是 $2^{26} * 4$ 字节，也就是 `268,435,456` 字节或 `256 MB`。

- **Summary:** Given a **Jump** instruction

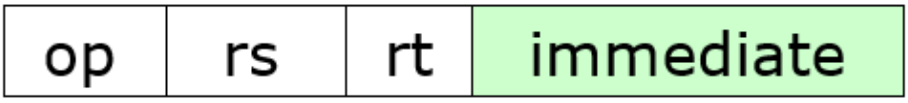


5.8 Addressing Modes

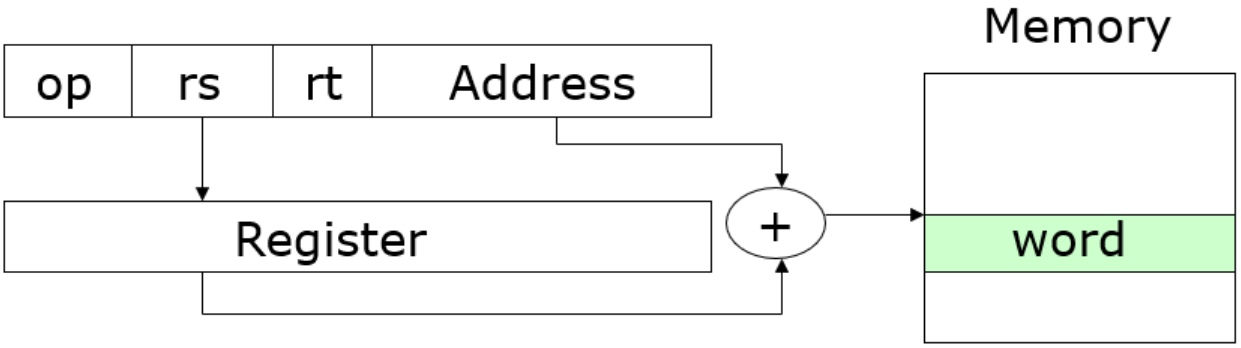
- **Register addressing**: operand is a register



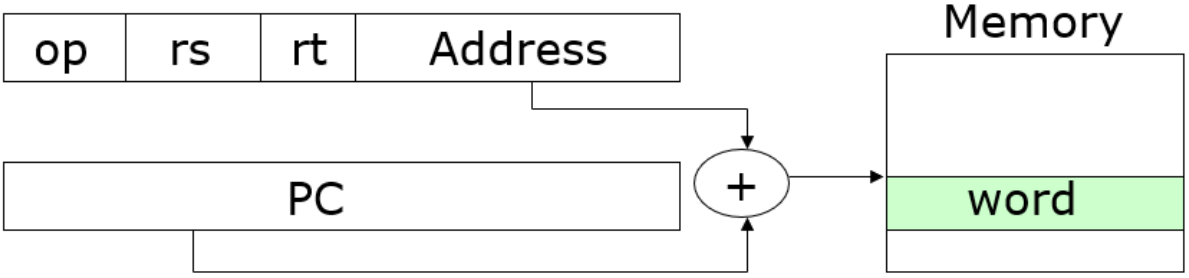
- Immediate addressing: operand is a constant within the instruction itself (`addi` , `andi` , `ori` , `slti`)



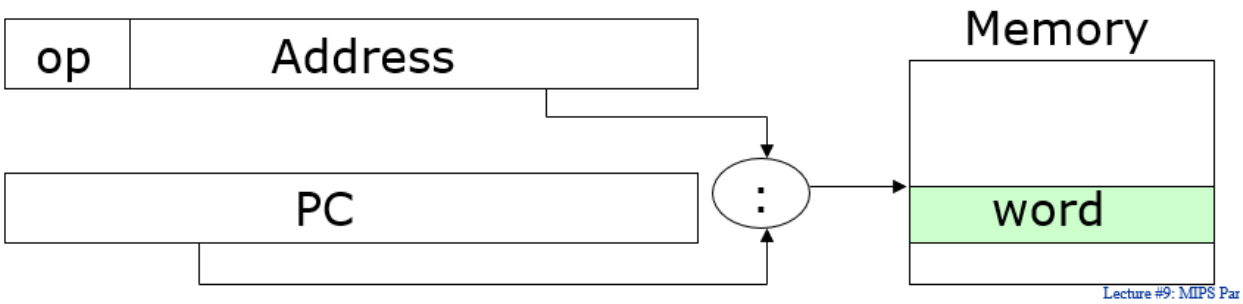
- Base addressing (displacement addressing): operand is at the memory location whose address is sum of a register and a constant in the instruction (`lw` , `sw`)



- 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
Conditional branch	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	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

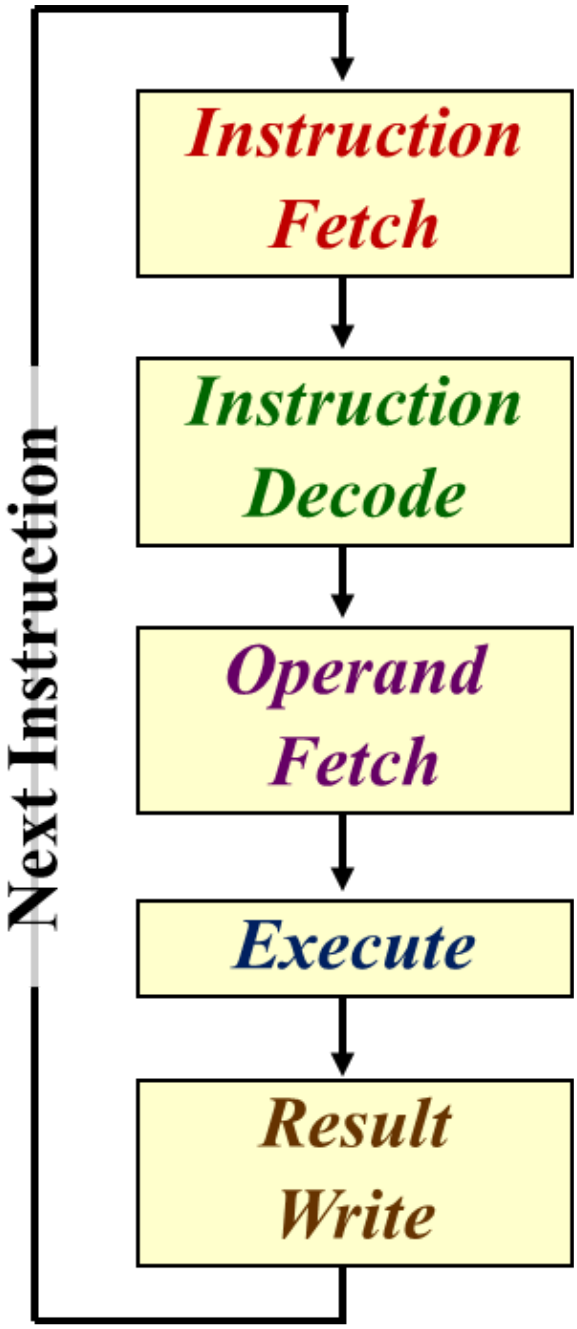
- Two major components for a processor:
  1. Datapath
     - Collection of components that process data
     - Performs the arithmetic, logical and memory operations
  2. Control
     - Tells the datapath, memory and I/O devices what to do according to program instructions

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*

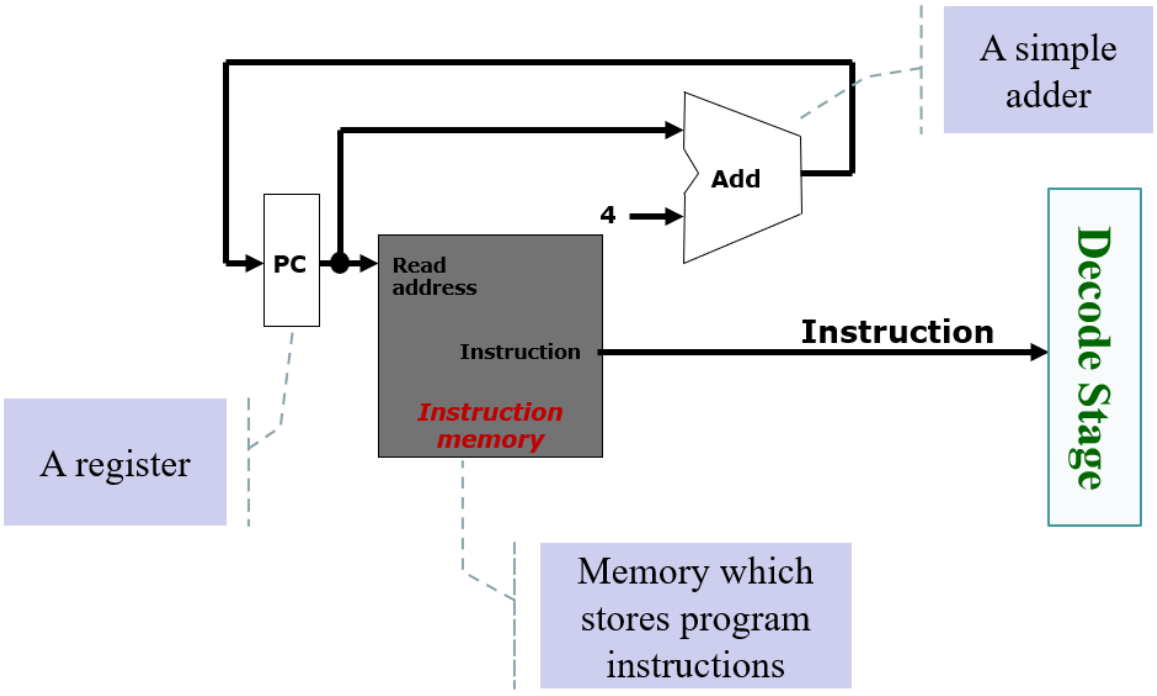
- `inst.` = instruction

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

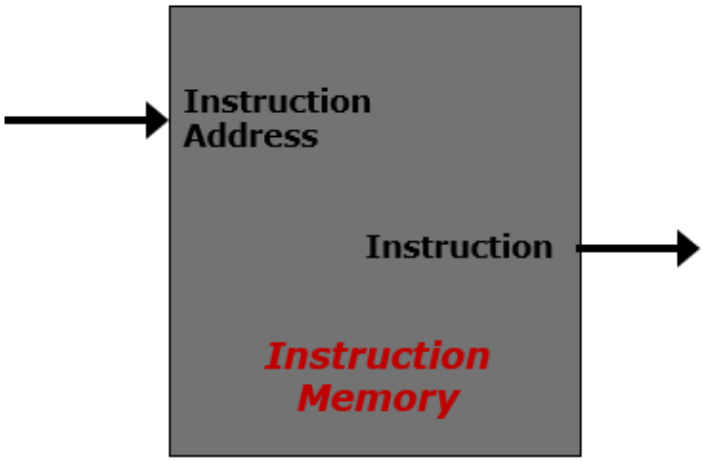
6.5.1 Fetch Stage

- 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

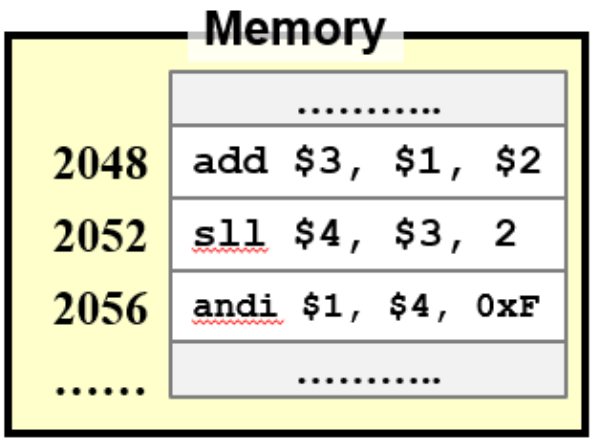


Element: Instruction Memory

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



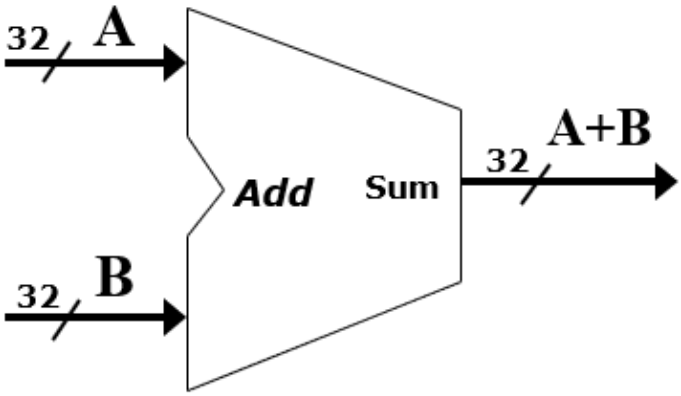
- Supply instruction given the address
  - Given instruction address M as input, the memory outputs the content at address M
  - Conceptual diagram of the memory layout is given on the below



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。

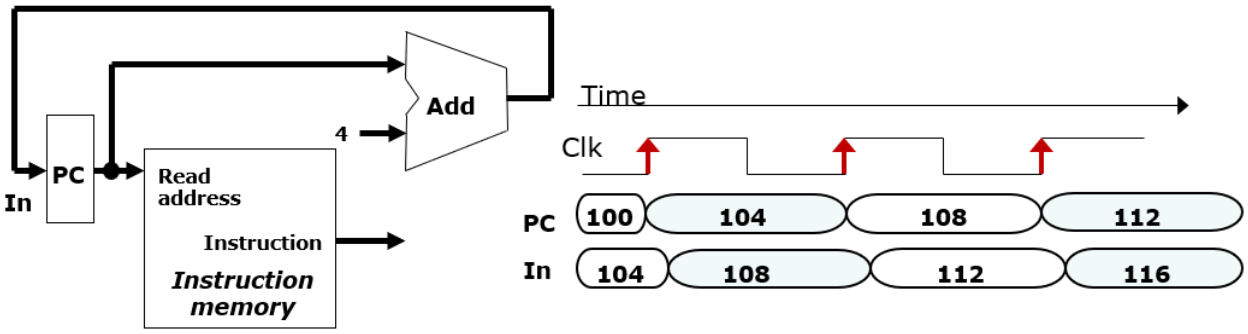
Element: Adder

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



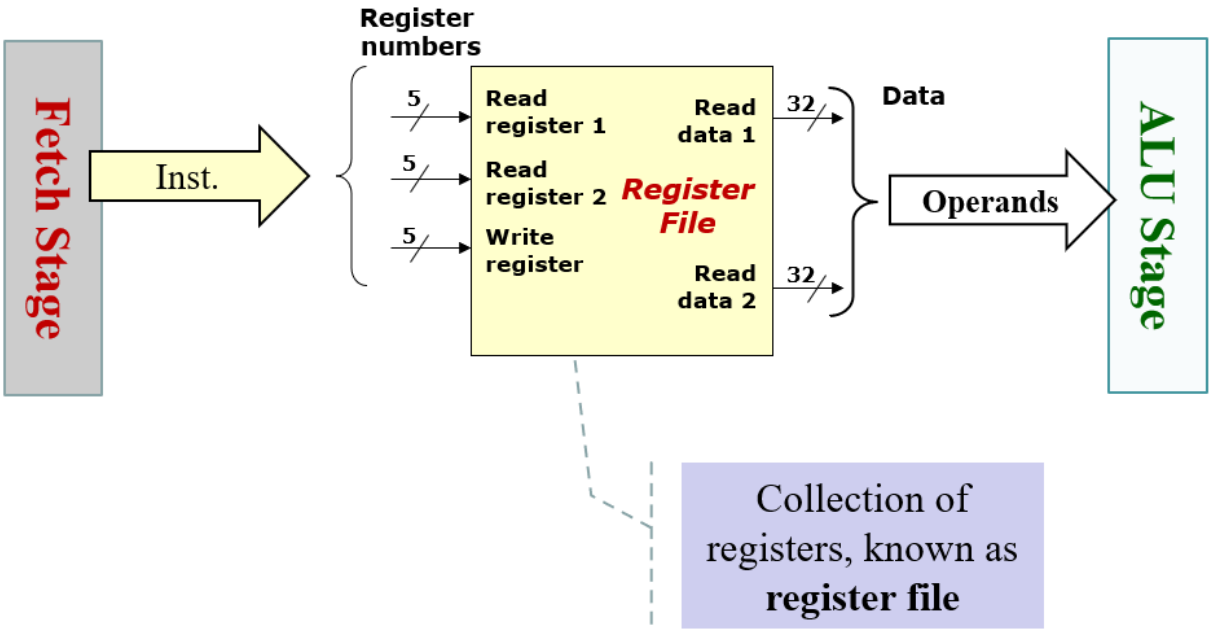
The idea of clocking

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



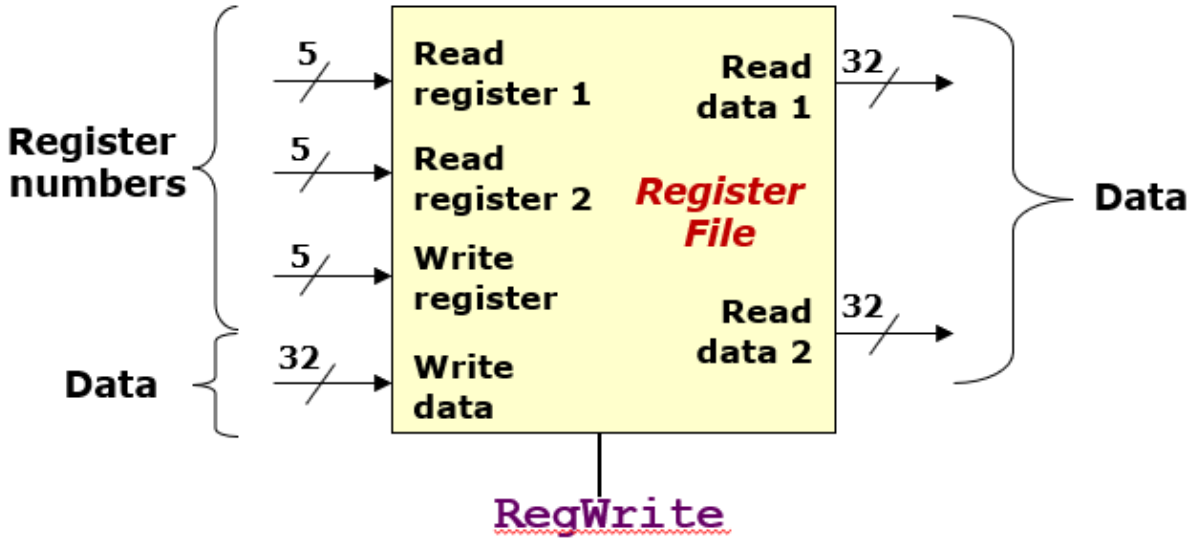
6.5.2 Decode Stage

- 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

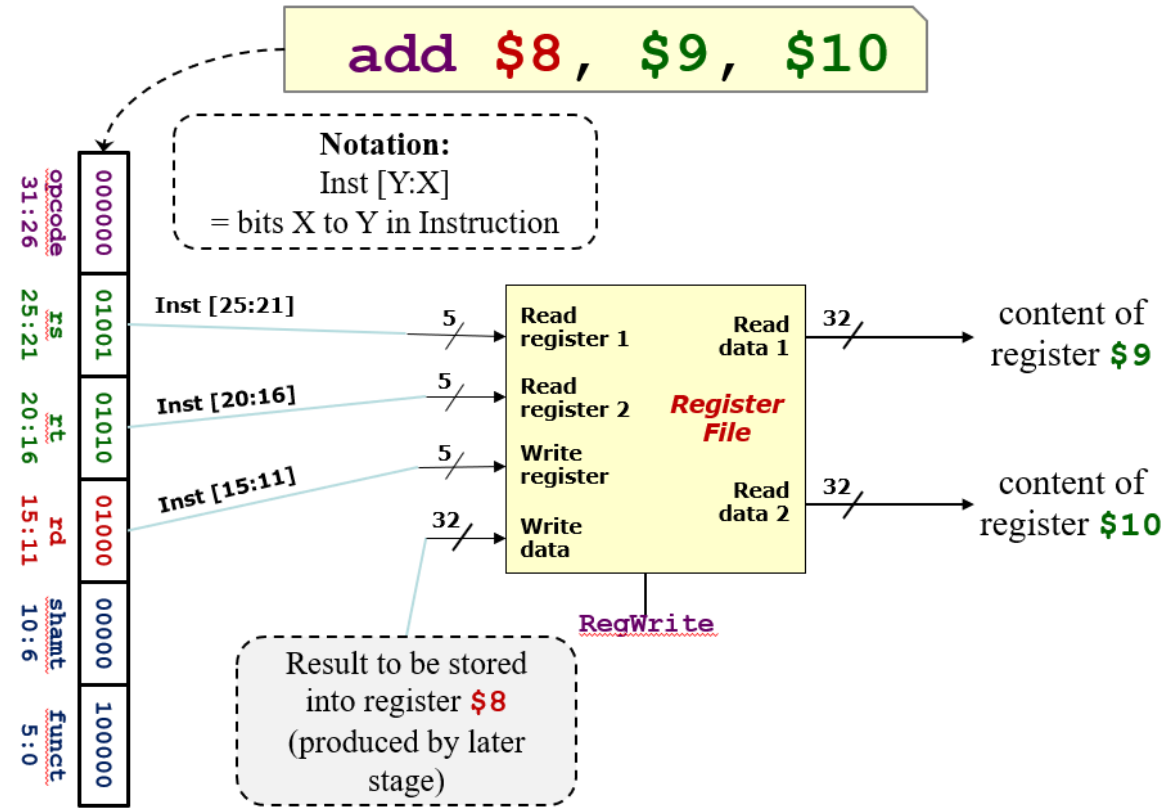


Element: Register File

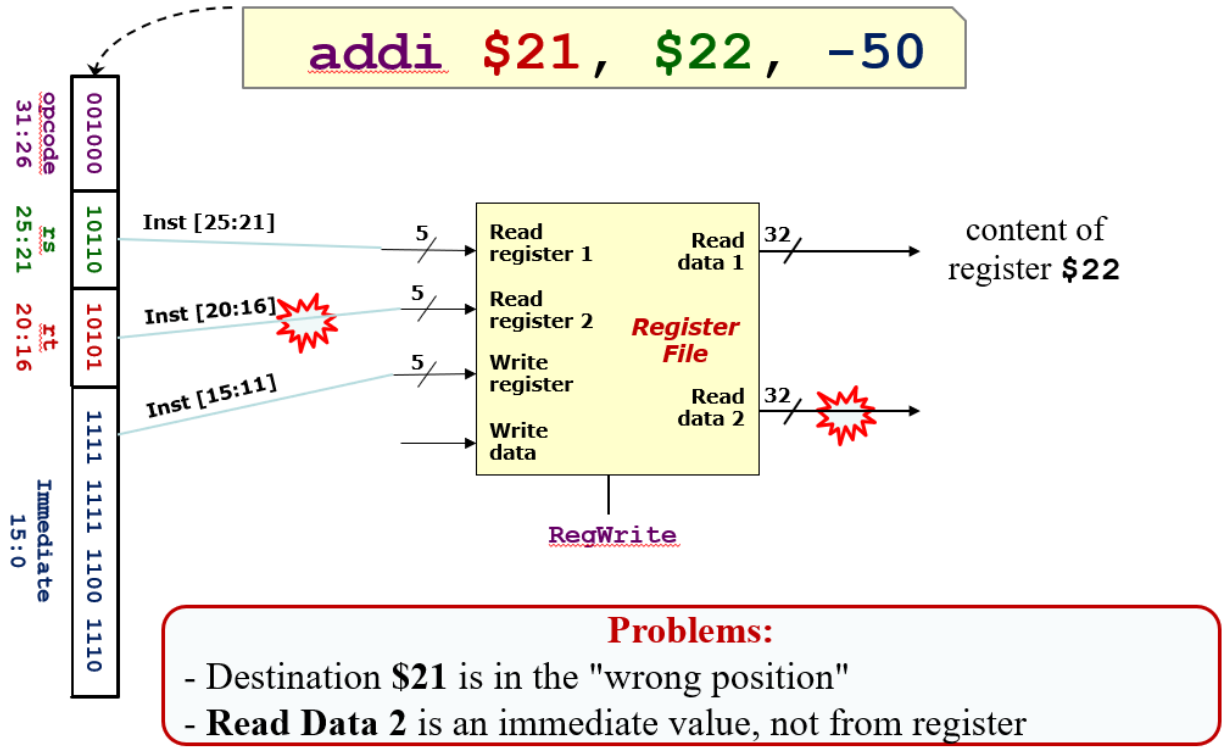
- A collection of 32 registers
  - Each 32-bit wide; can be read/written by specifying register number
  - Read at most two register per instruction
  - Write at most one register per instruction
- `RegWrite` is a control signal to indicate:
  - Writing of register
  - 1 (True) = Write, 0 (False) = No Write



R-Format Instruction

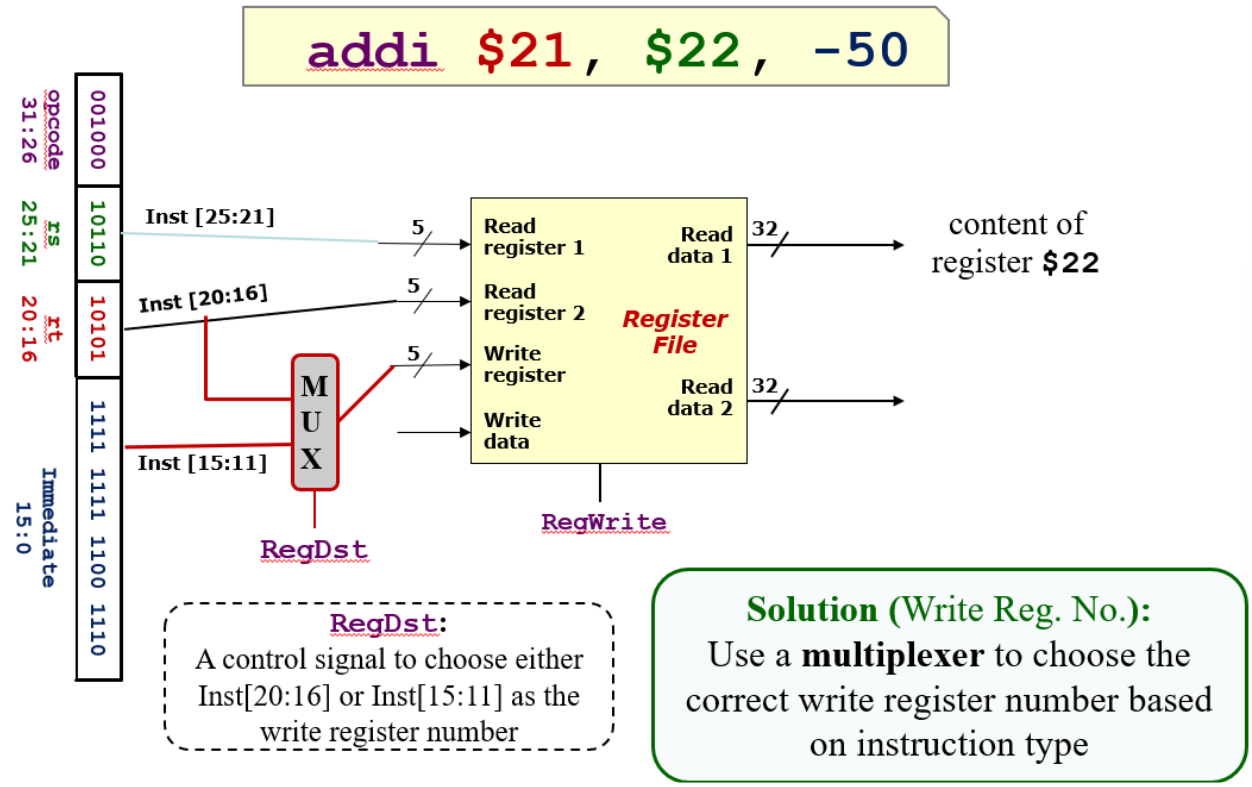


I-Format Instruction



Problems:

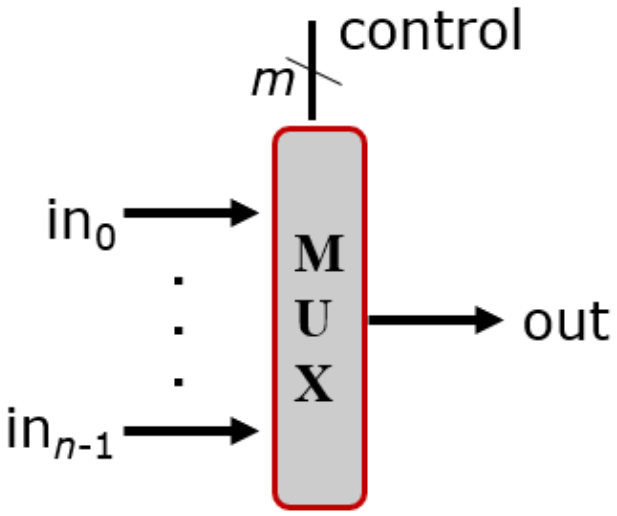
- Destination **$21** is in the "wrong position"
- **Read Data 2** is an immediate value, not from register



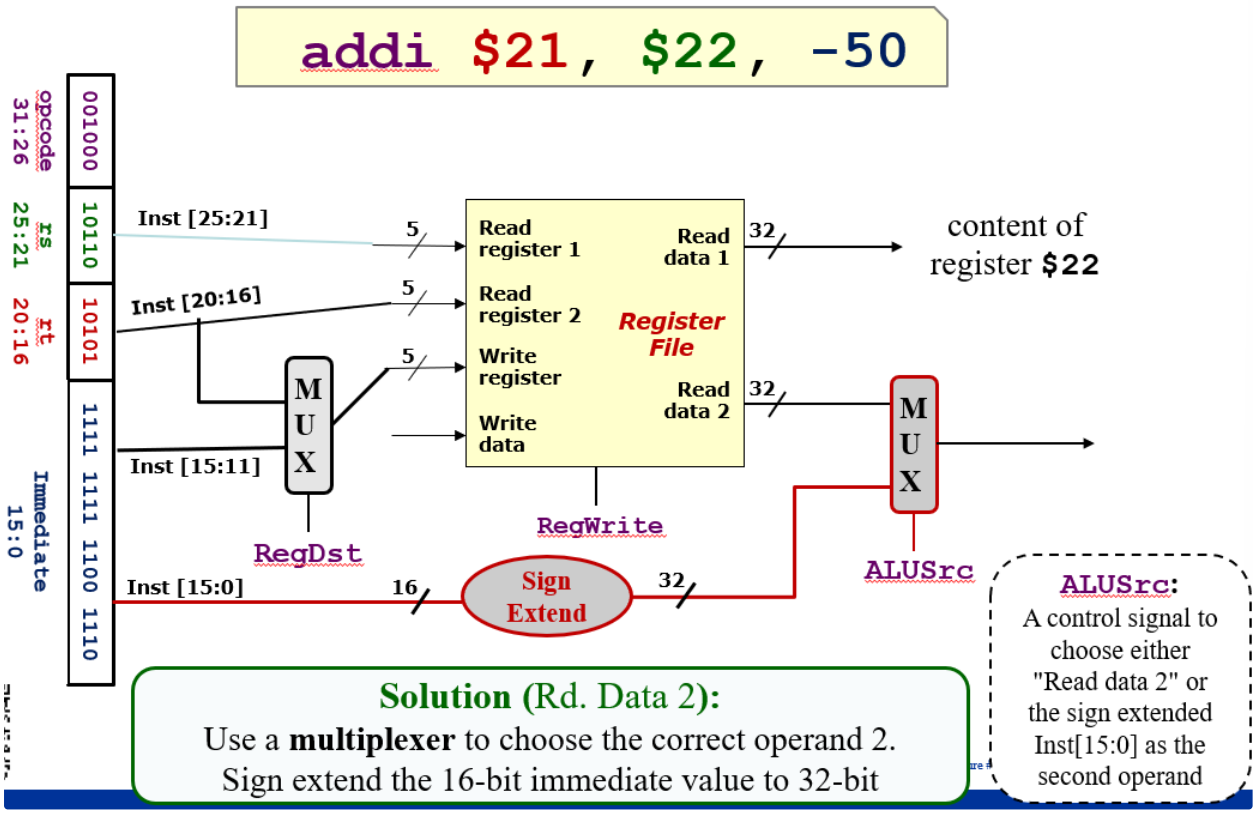
1. **不同的指令格式**：MIPS指令集包括R-format、I-format和J-format等不同格式的指令。这些指令格式中，寄存器的编号和位置可能会不同。
2. **写入寄存器的选择**：在R-format指令中，通常使用 `rd` 字段（也就是 `Inst[15:11]` ）来指定结果的写入寄存器。但是，在I-format指令（如 `addi` ）中，结果是写入 `rt` 字段指定的寄存器，也就是 `Inst[20:16]` 。
3. **multiplexer的作用**：为了能够处理这种不同，我们需要一个选择器，即 `multiplexer` 。根据指令的类型（R-format还是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。

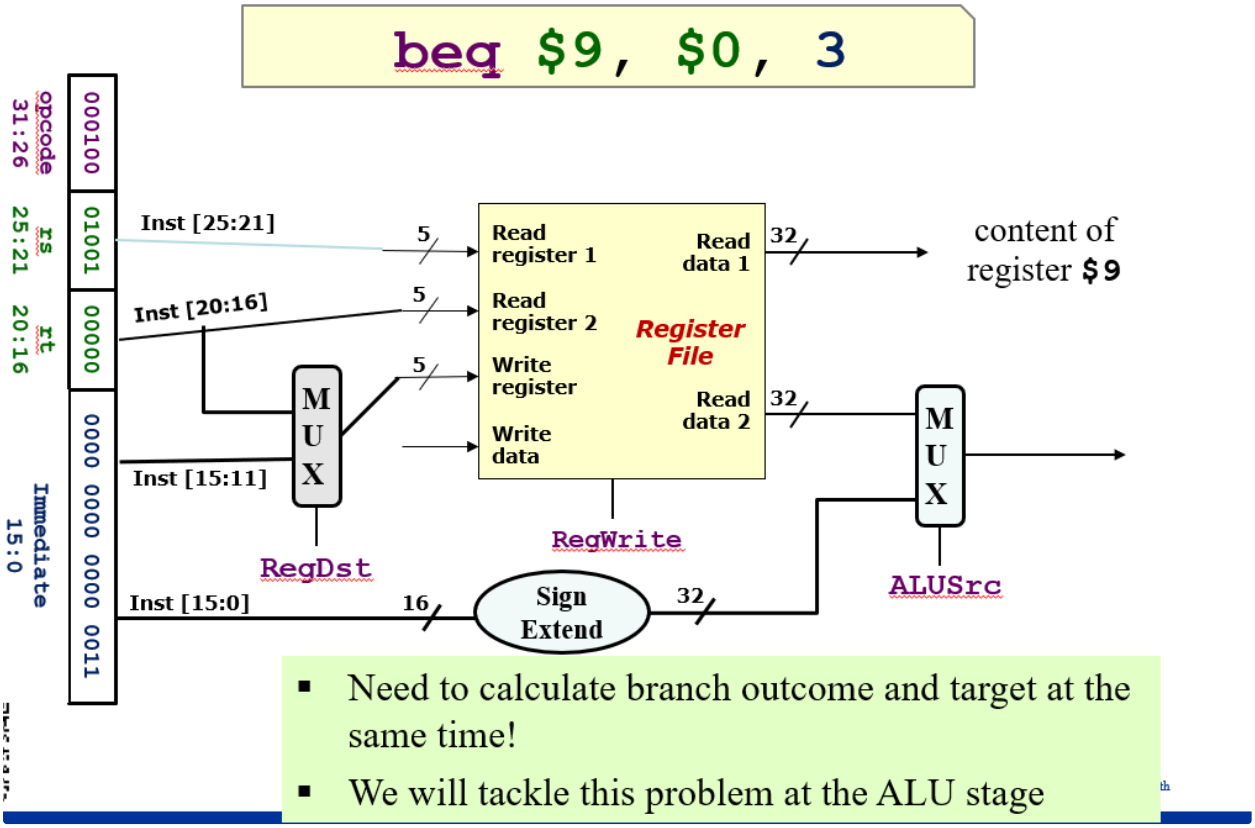
首先在左侧的输入部分，由于I format指令少一个寄存器的输入，所以先加入一个MUX将rt部分和原先属于rd部分的寄存器输入，根据指令的类型选择输出一个到decoder中。最后，对immediate的后半部分（16位）加长到32位后连到data read 2后面的MUX中。

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

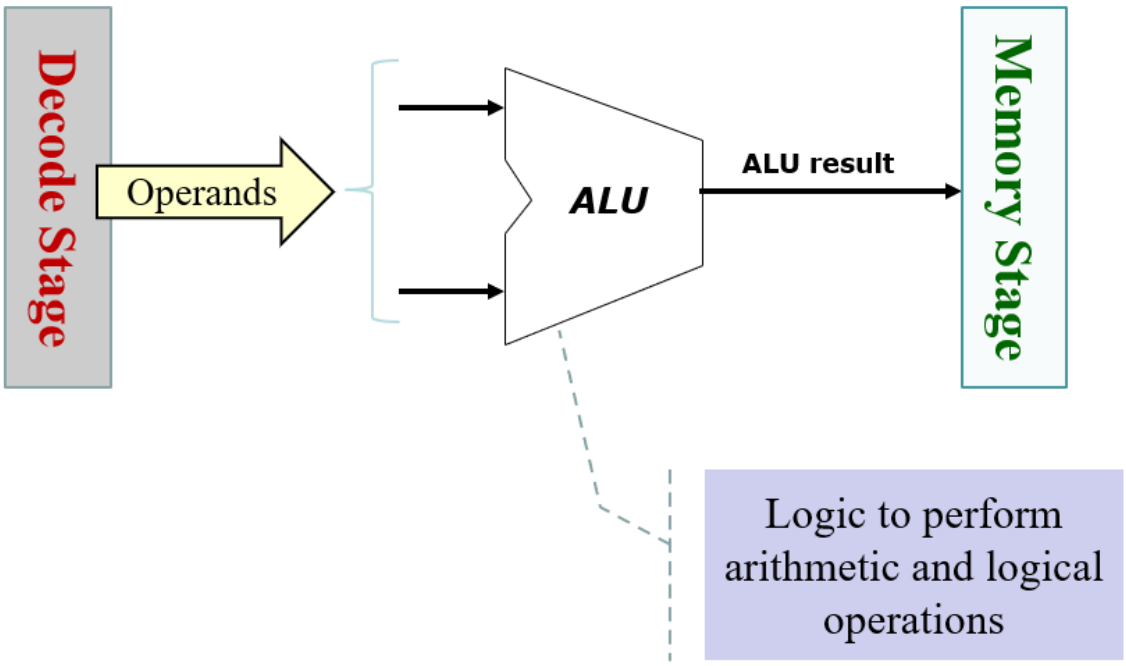
> 在右半部分，如果指令是I format的，那么左侧的MUX会输入rt，右侧的MUX会选择immediate。最终decode阶段输出的是一个register值和一个immediate值

Branch Instruction



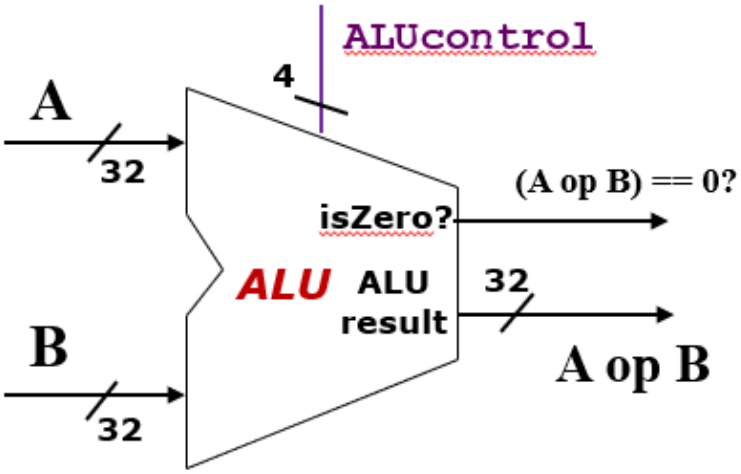
6.5.3 ALU Stage

- 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



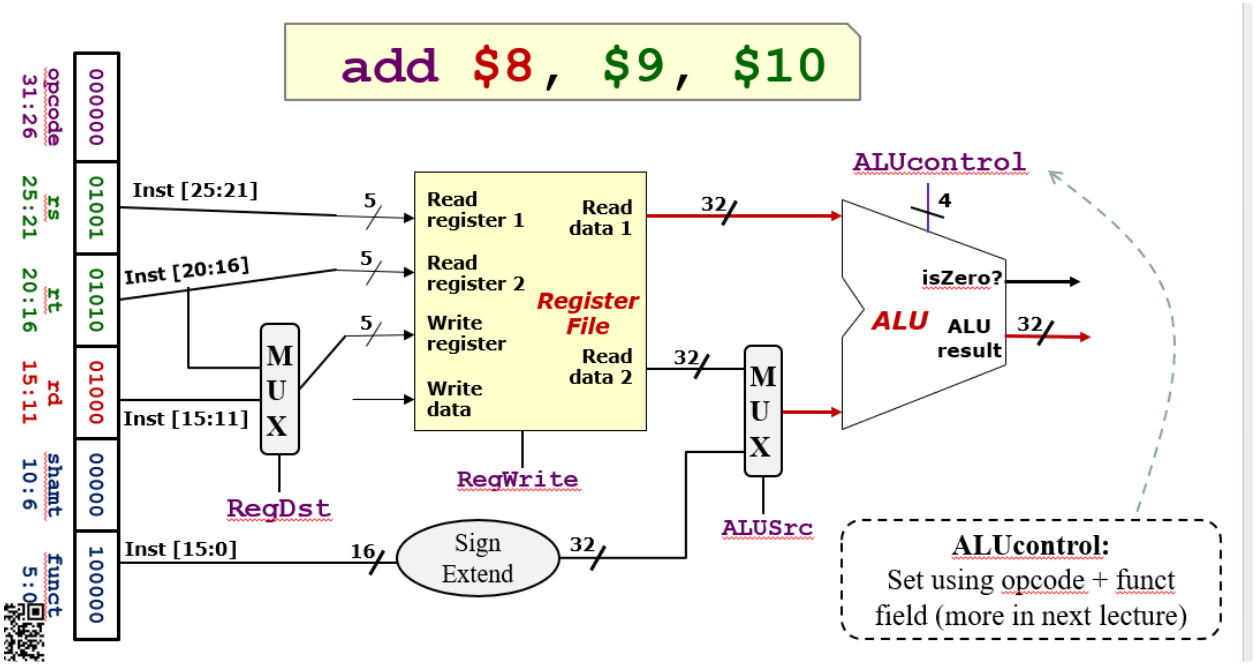
Element: Arithmetic Logic Unit

- 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



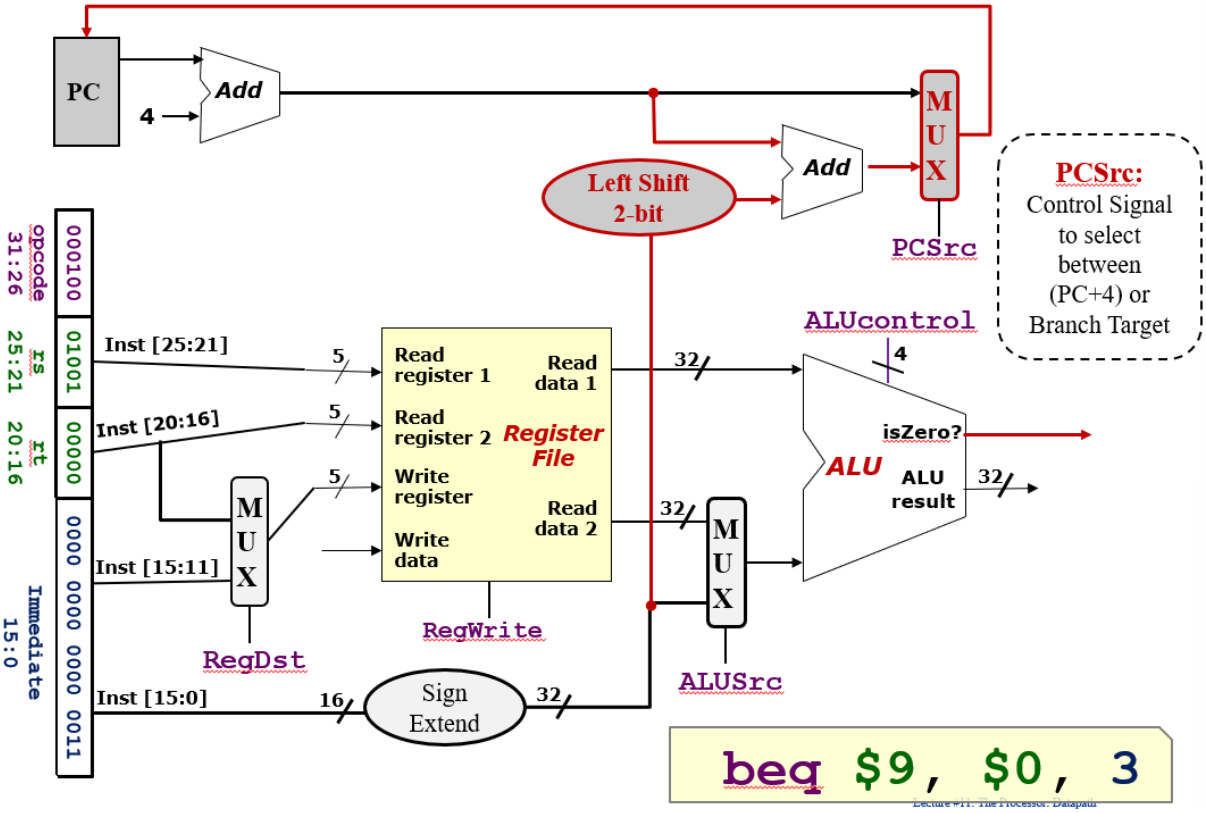
ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR

Decode & ALU Stage: Non-Branch Instructions



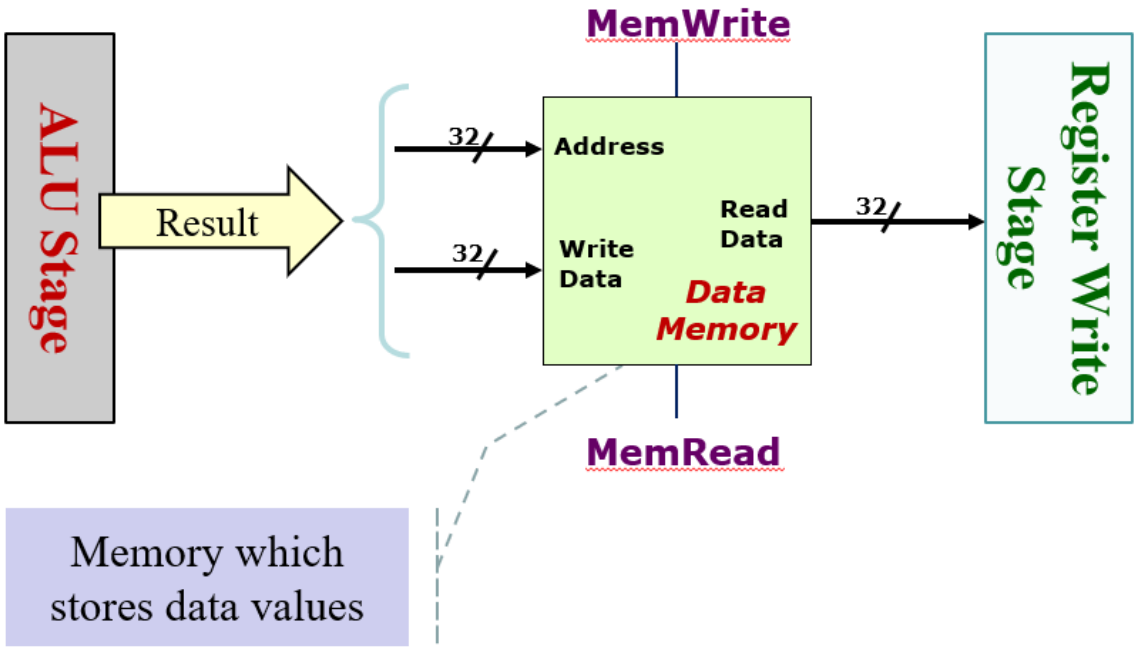
ALU Stage: Branch Instructions

- 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)



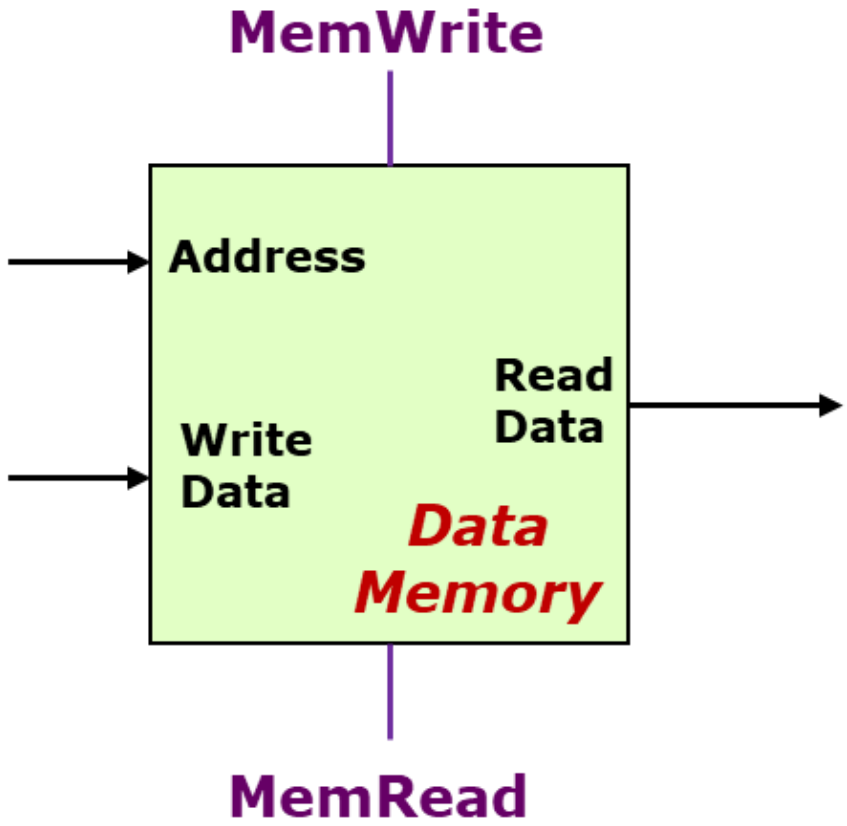
6.5.4 Memory Stage

- 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)



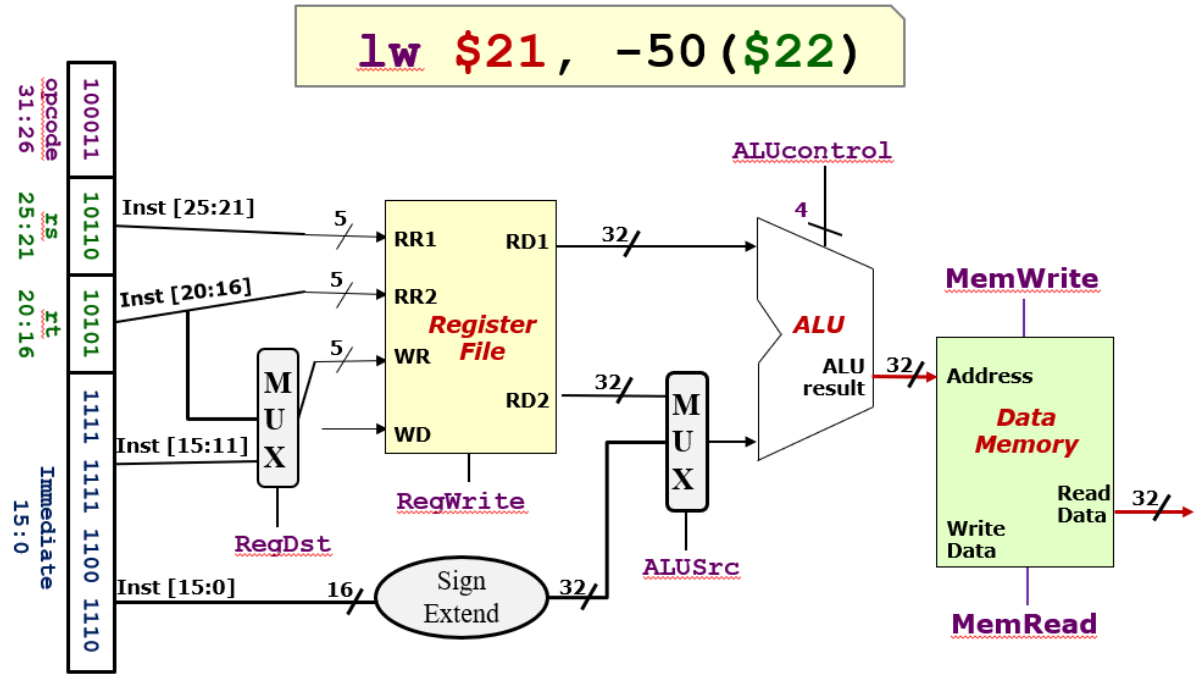
Element: Data Memory

- 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



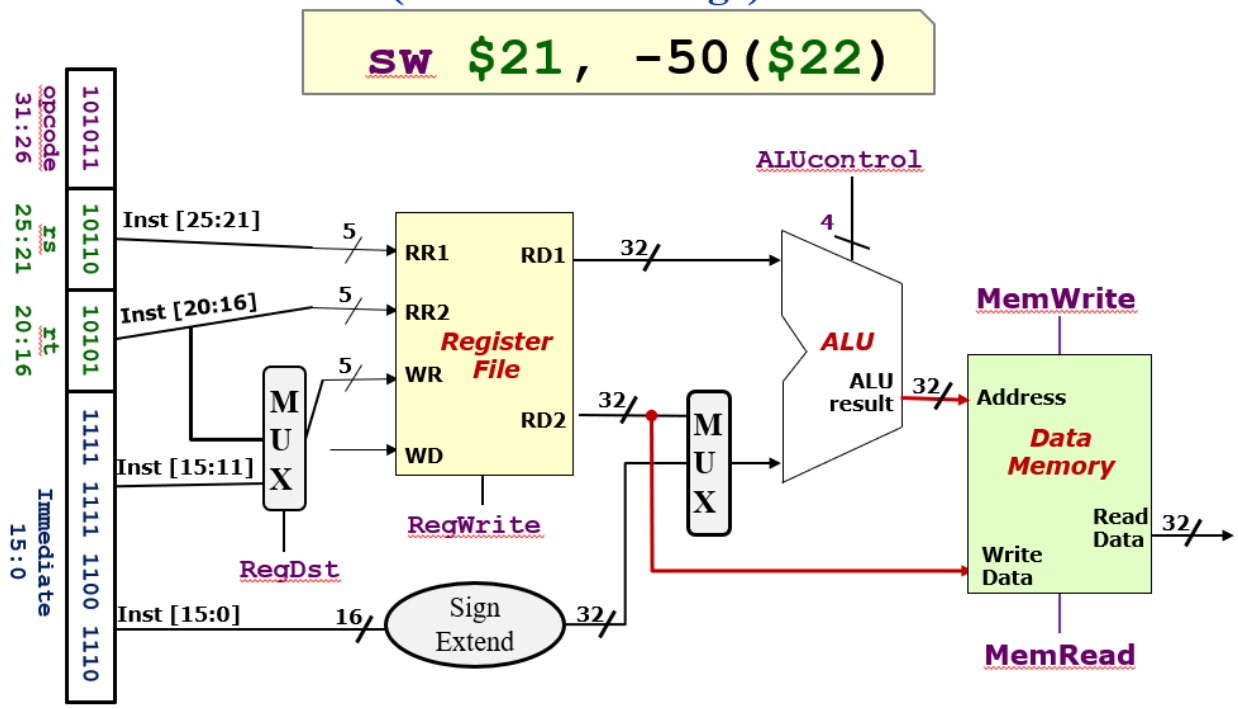
Load Instruction

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



Store Instruction

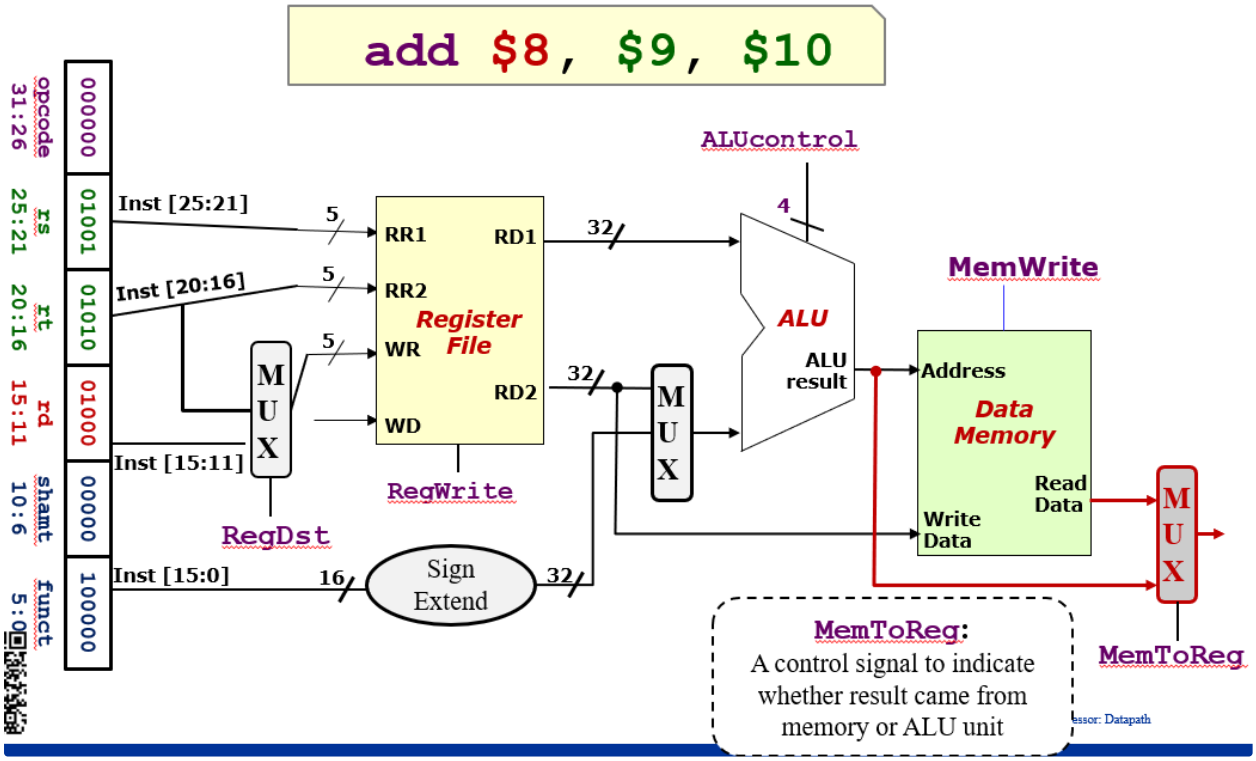
- **Need *Read Data 2* (from Decode stage) as the *Write Data***



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` 。

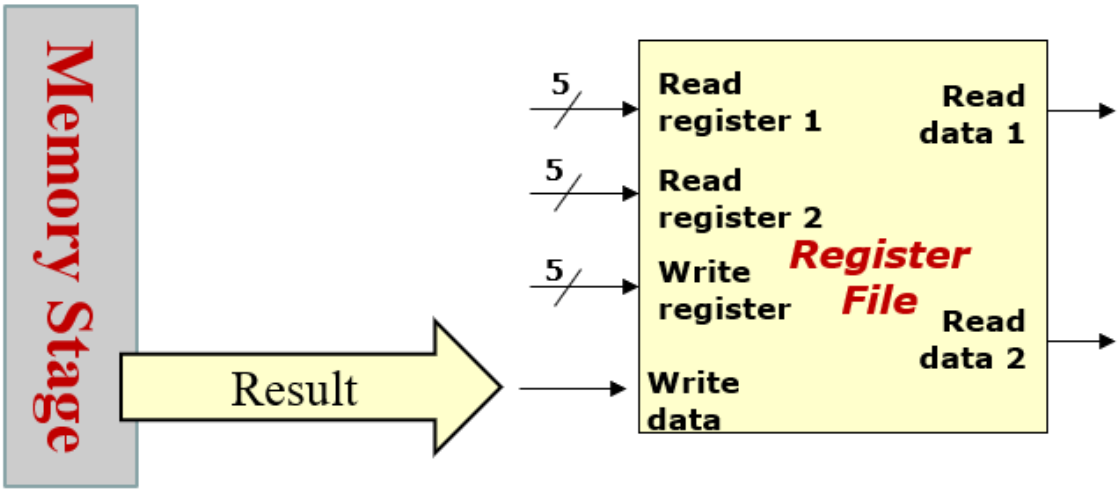
For Non-Memory Instruction (`add`)

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



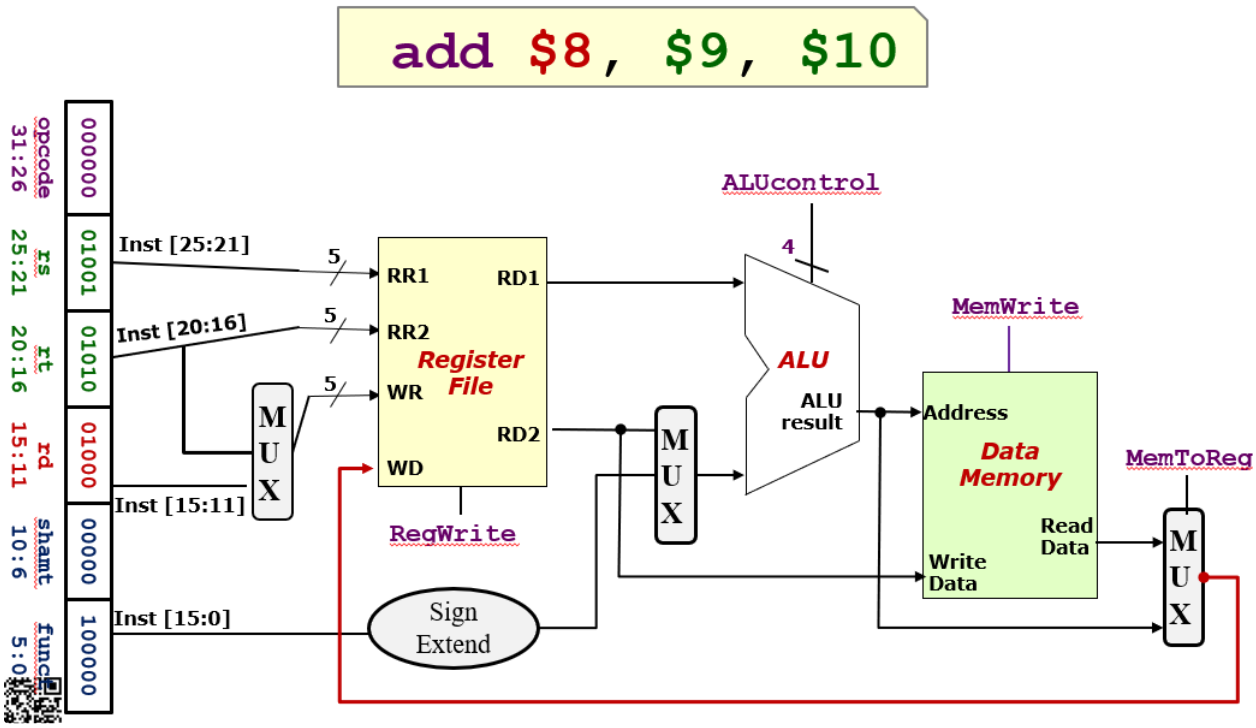
6.5.5 Register Write Stage

- 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

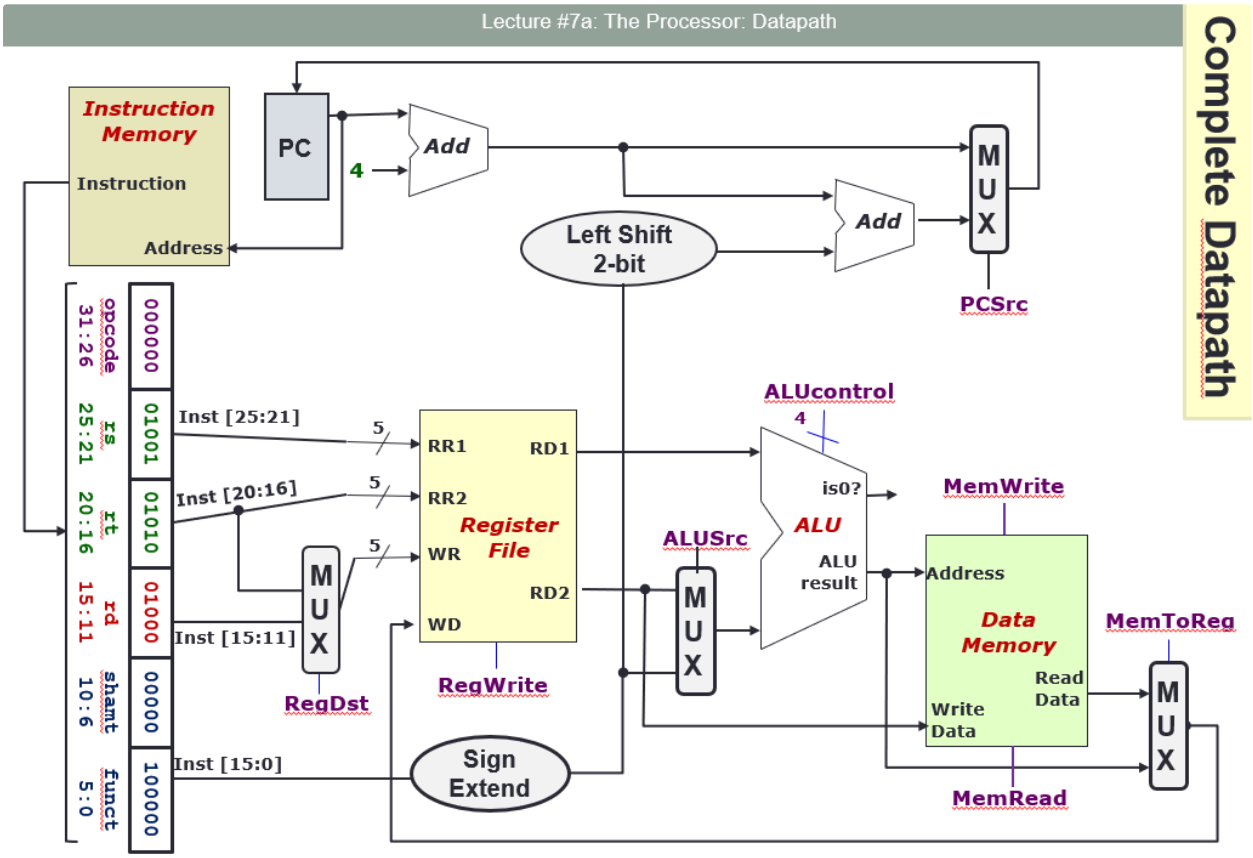


- Result Write stage has no additional element:
  - Basically just route the correct result into register file
  - The Write Register number is generated way back in the Decode Stage

Routing

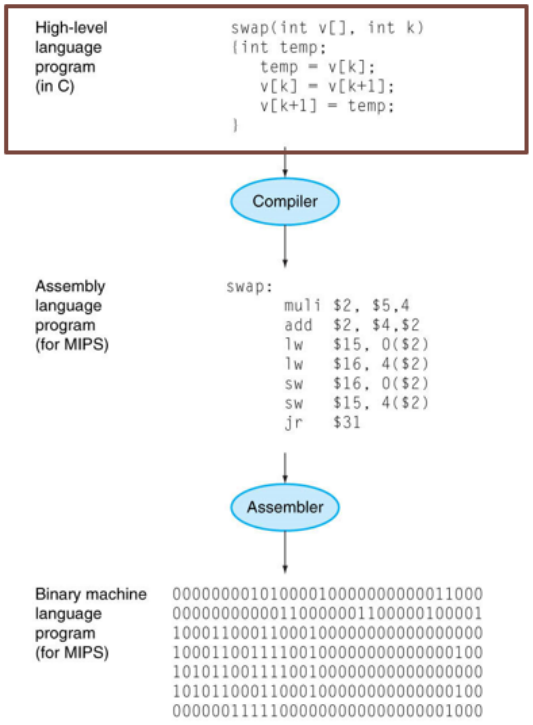


6.6 Complete Datapath



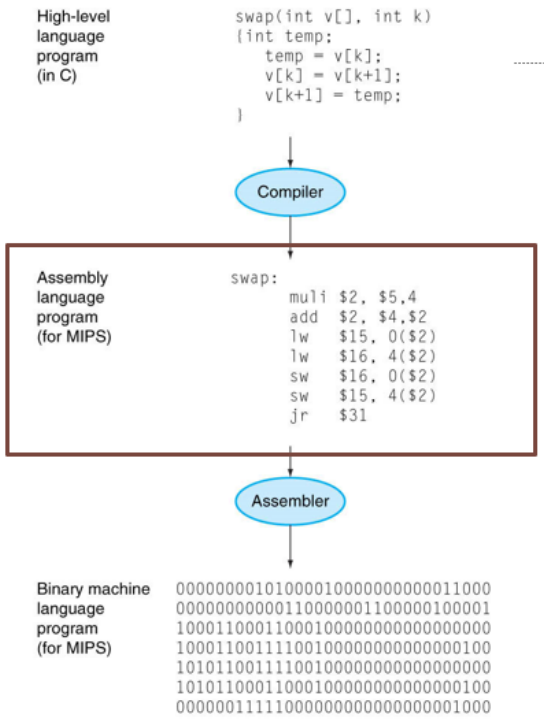
7 – The Processor: Datapath

7.1 Brief Recap



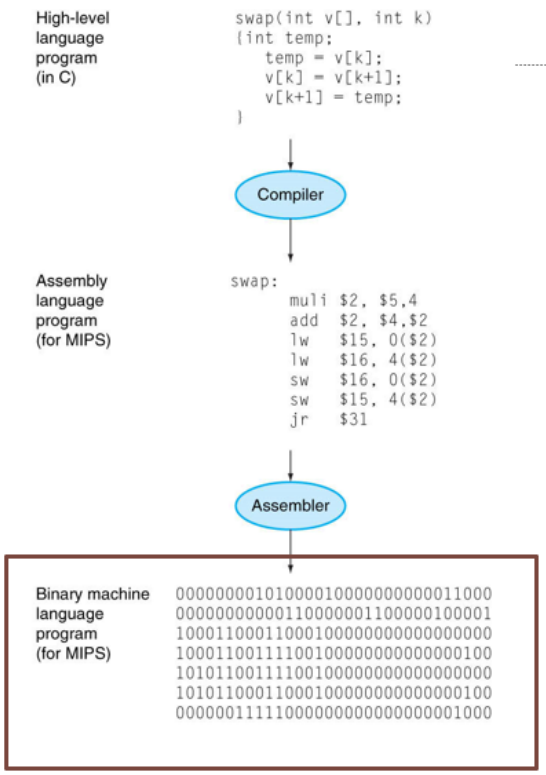
- Write program in high-level language (e.g., **C**)

```
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:
```



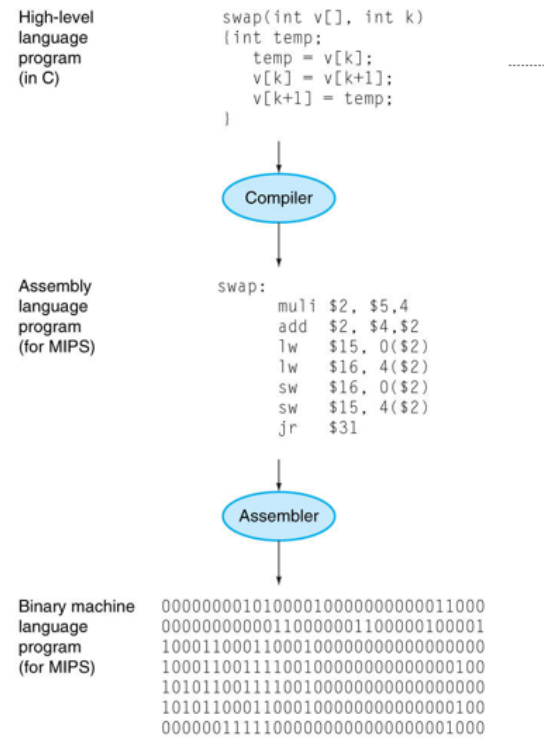
➤ Assembler translates to machine code (i.e., **binaries**)

```
0001 0010 0000 0000
0000 0000 0000 0011

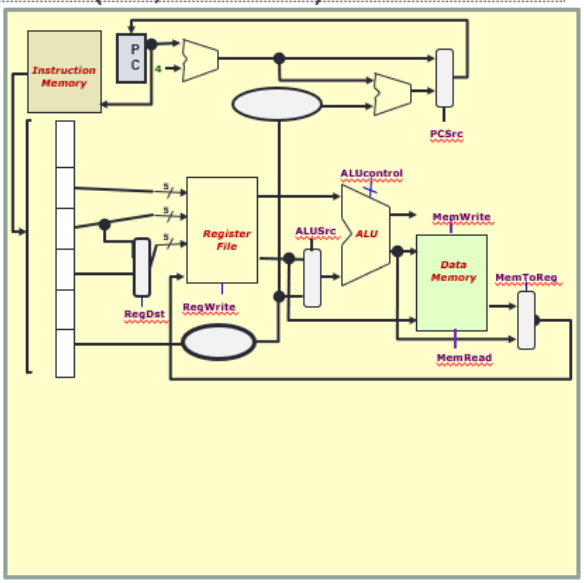
1000 1110 0010 1000
0000 0000 0000 0100

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

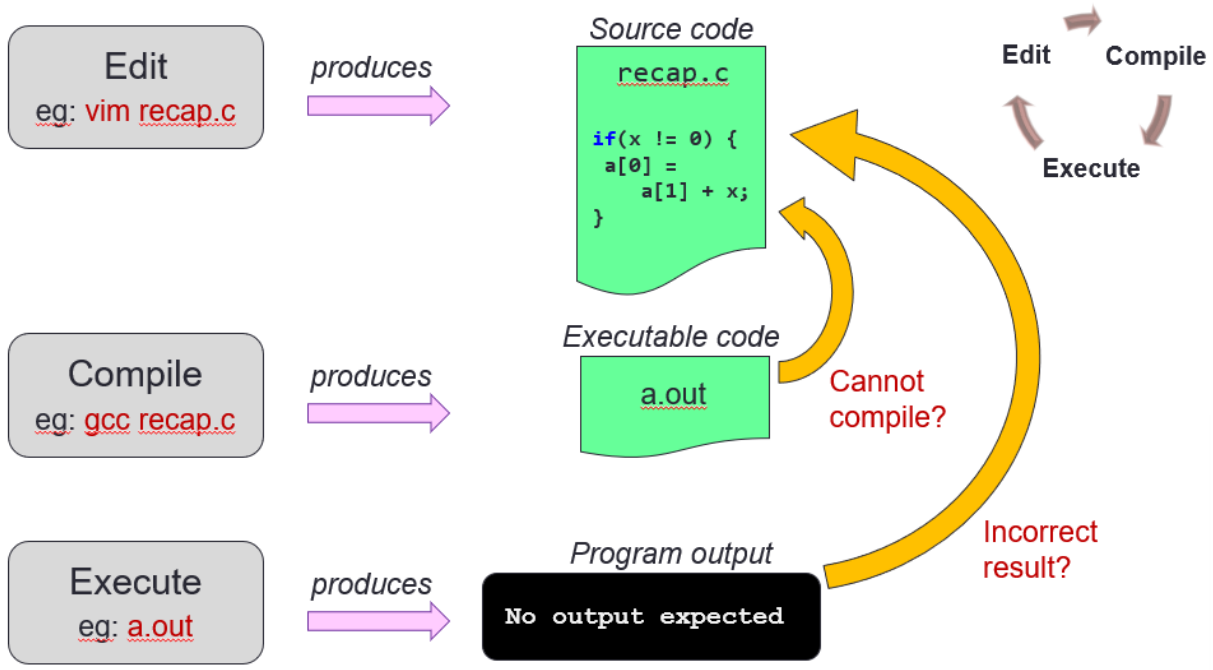
- We play the role of Programmer, Compiler, Assembler, and Processor
  - Program:

```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

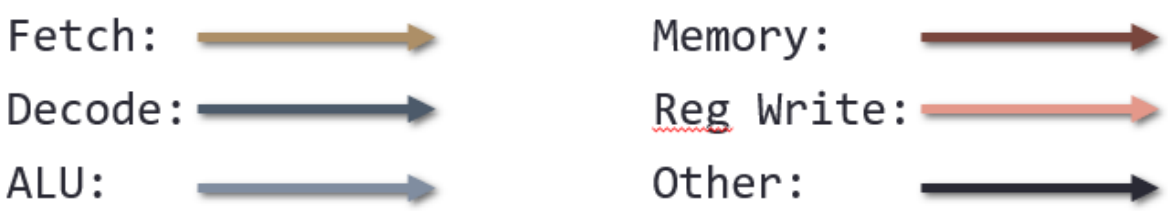
   - Convert to binary

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:
```

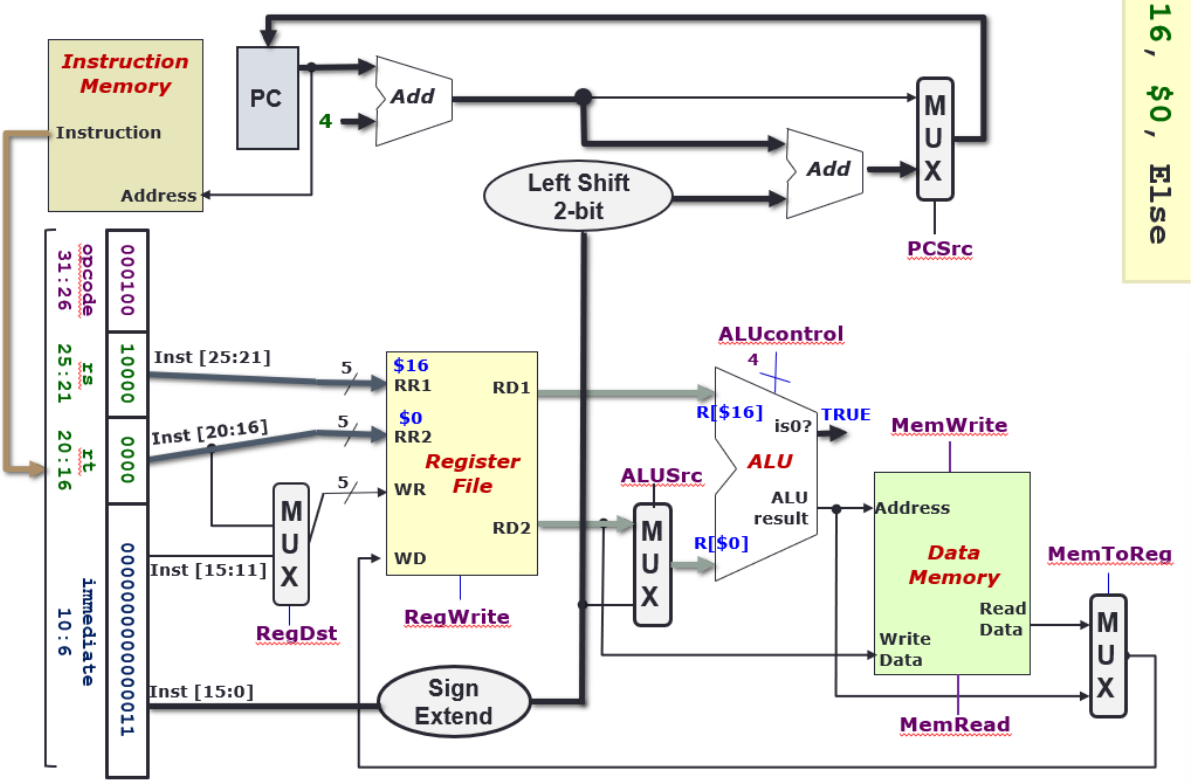
7.2.4 Execution (Datapath)

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



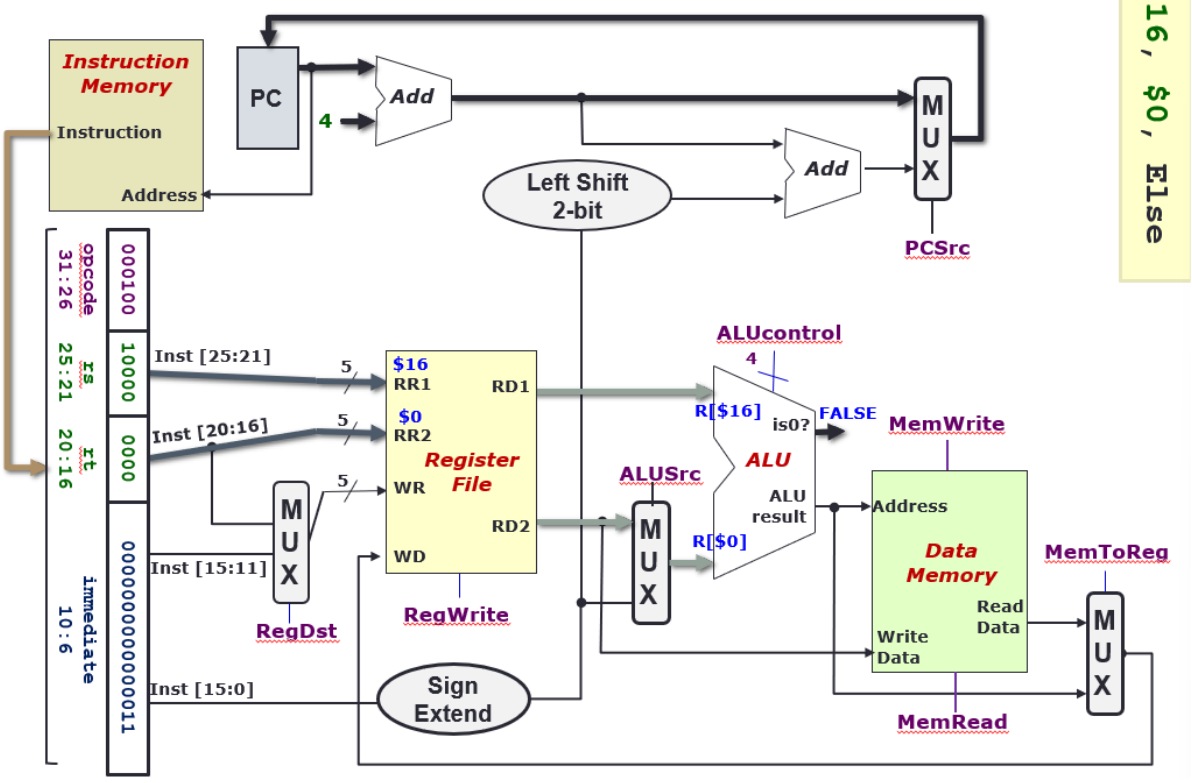
beq $16, $0, Else

- Assume $16 == $0



beq $16, $0, Else

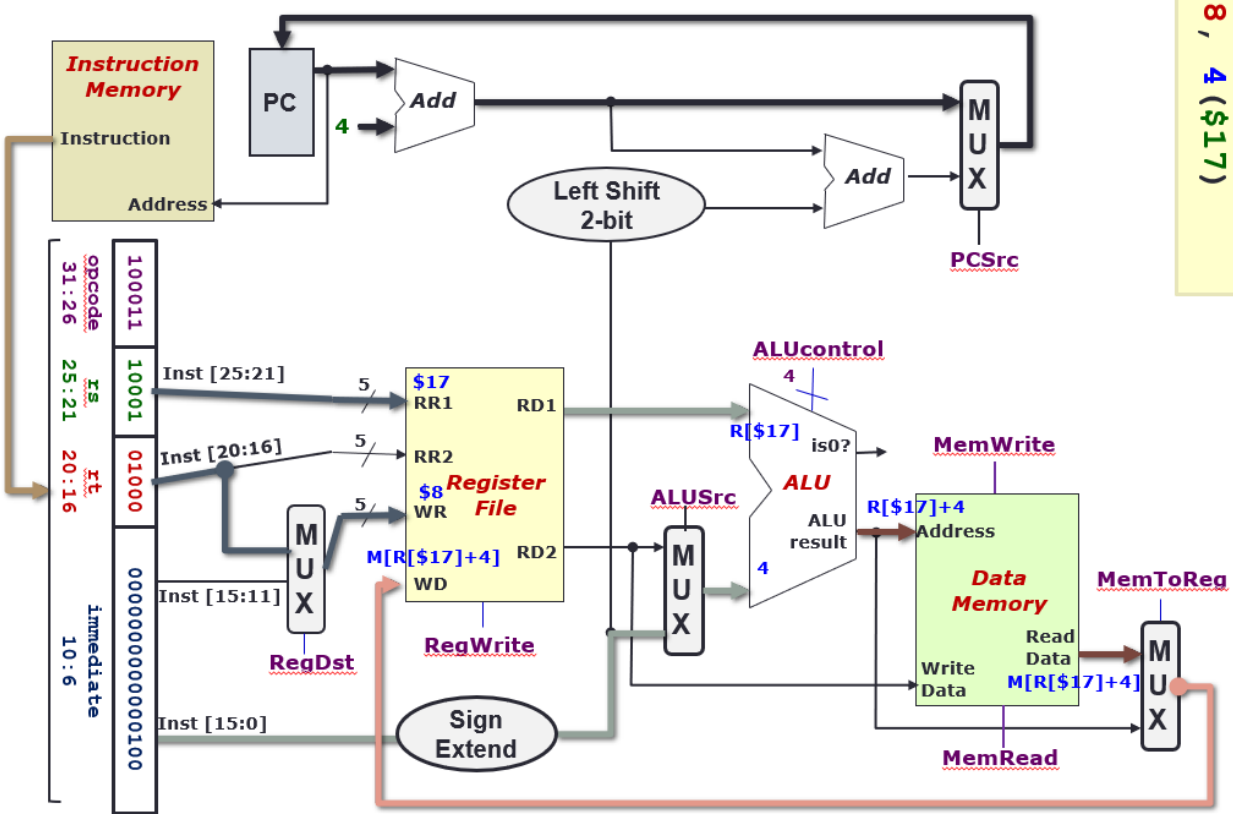
- Assume $16 != $0



Lecture #7a: The Processor: Datapath

lw $8, 4($17)

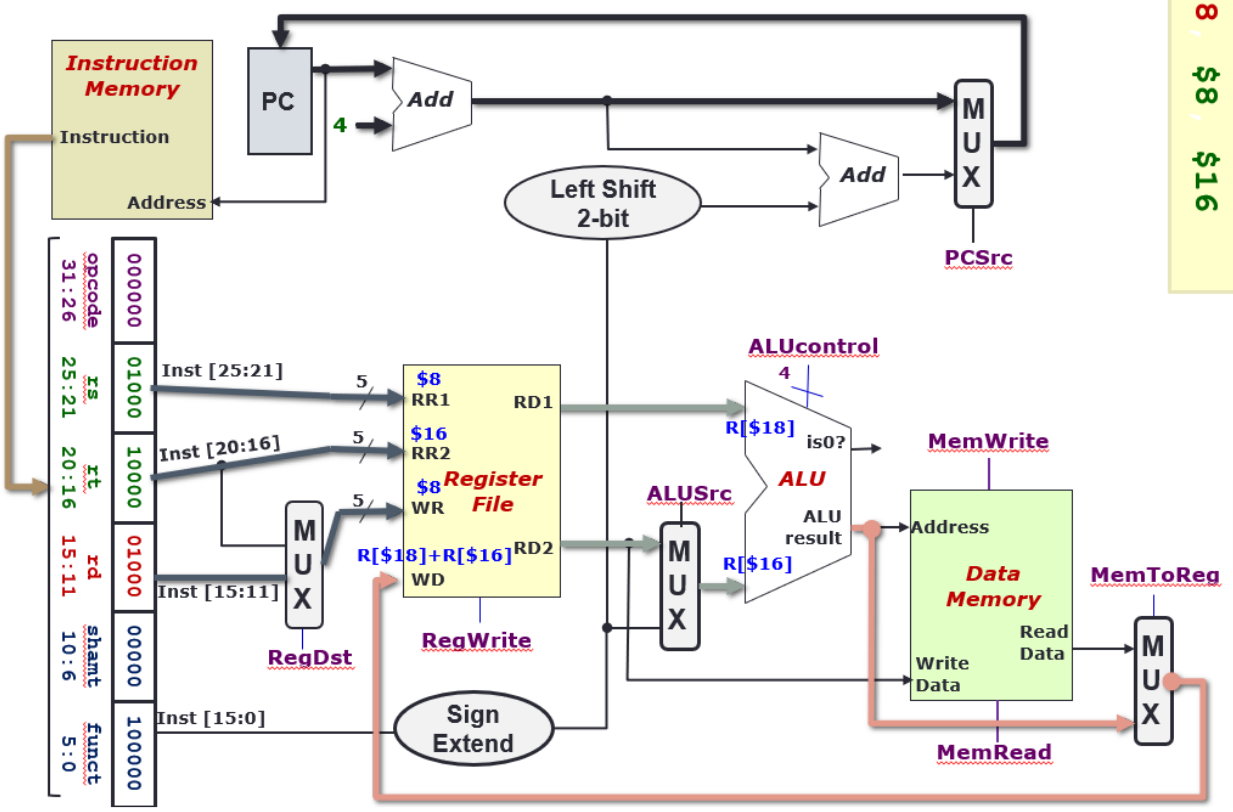
- Assume $16 != $0

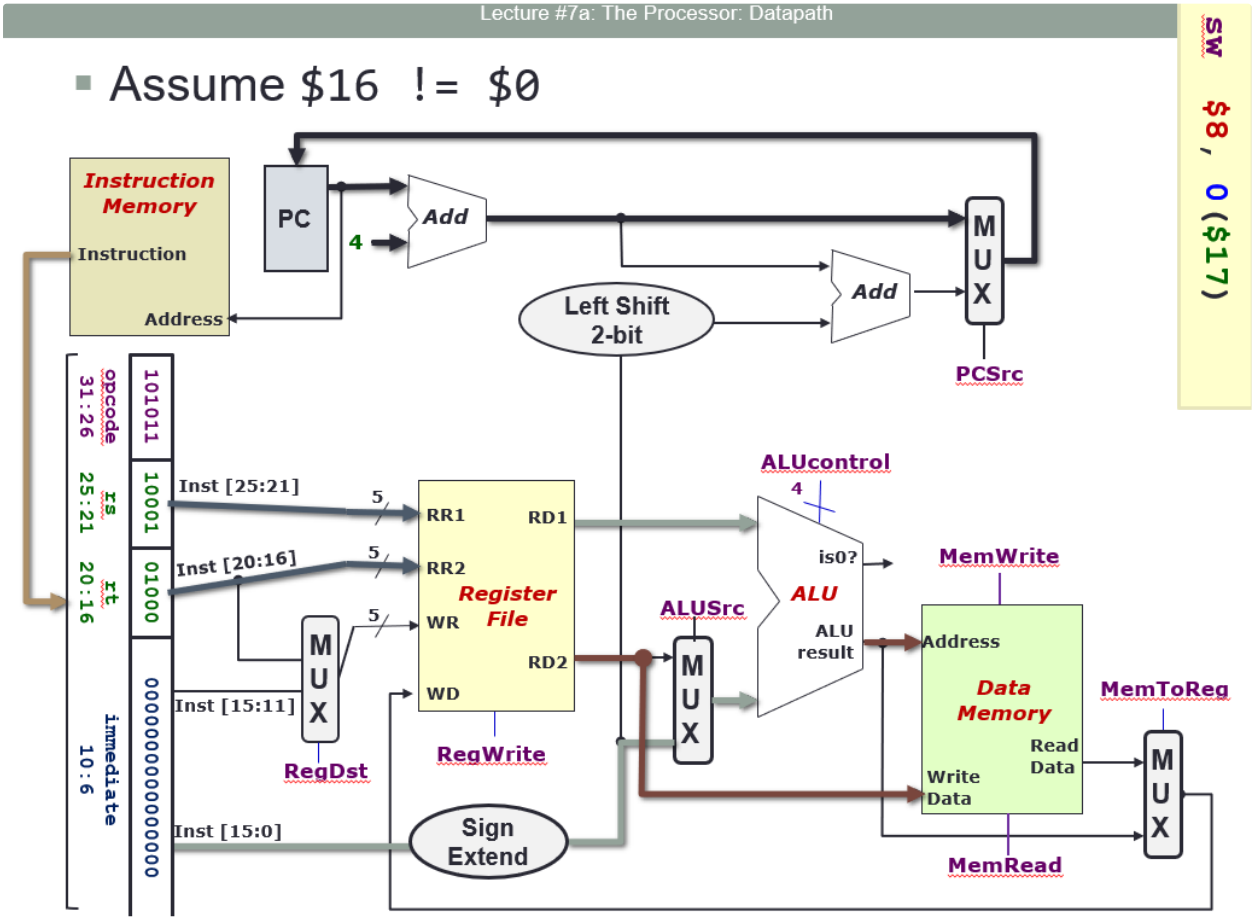


Lecture #7a: The Processor: Datapath

add $8, $8, $16

- Assume $16 != $0





8 - Processor: Control

8.1 Identified Control Signals

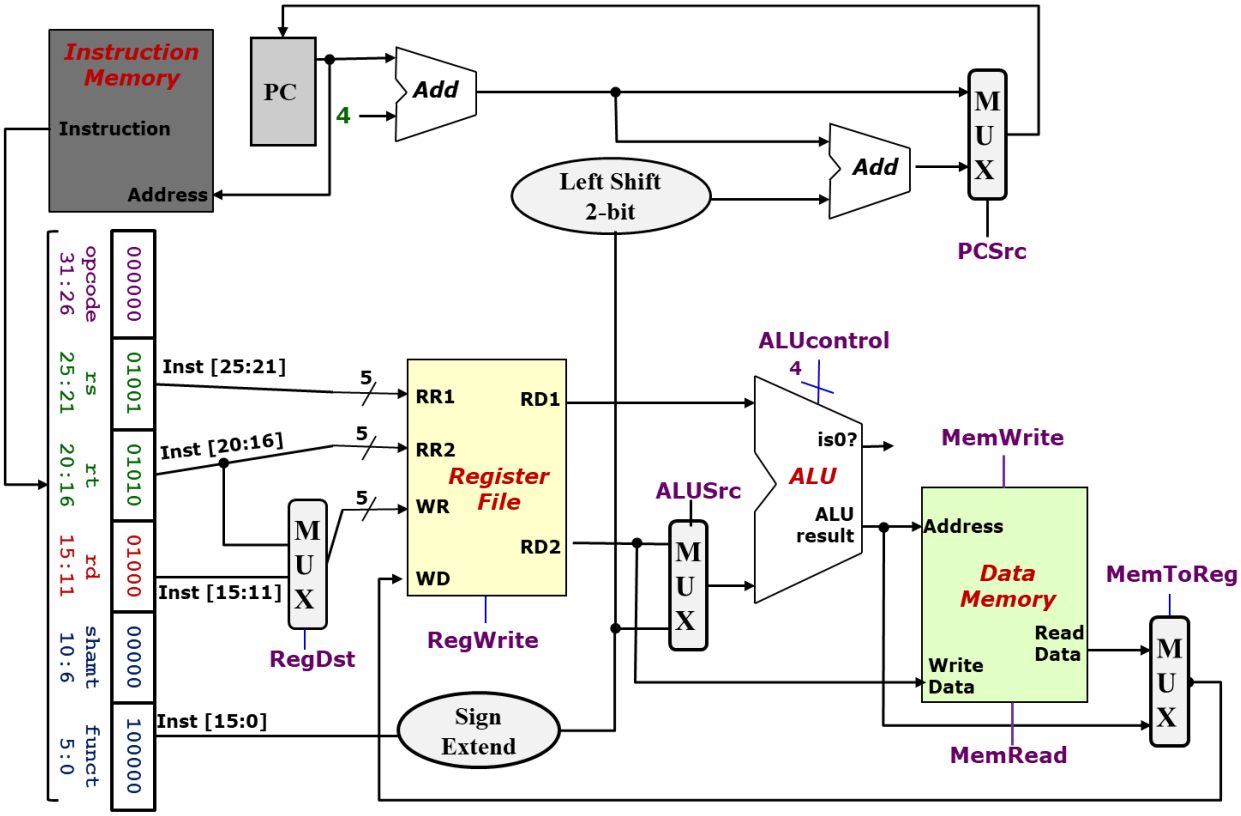
Control Signal	Execution Stage	Purpose
RegDst	Decode/Operand Fetch	Select the destination register number
RegWrite	Decode/Operand Fetch RegWrite	Enable writing of register
ALUSrc	ALU	Select the 2nd operand for ALU
ALUControl	ALU	Select the operation to be performed
MemRead / MemWrite	Memory	Enable reading/writing 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。

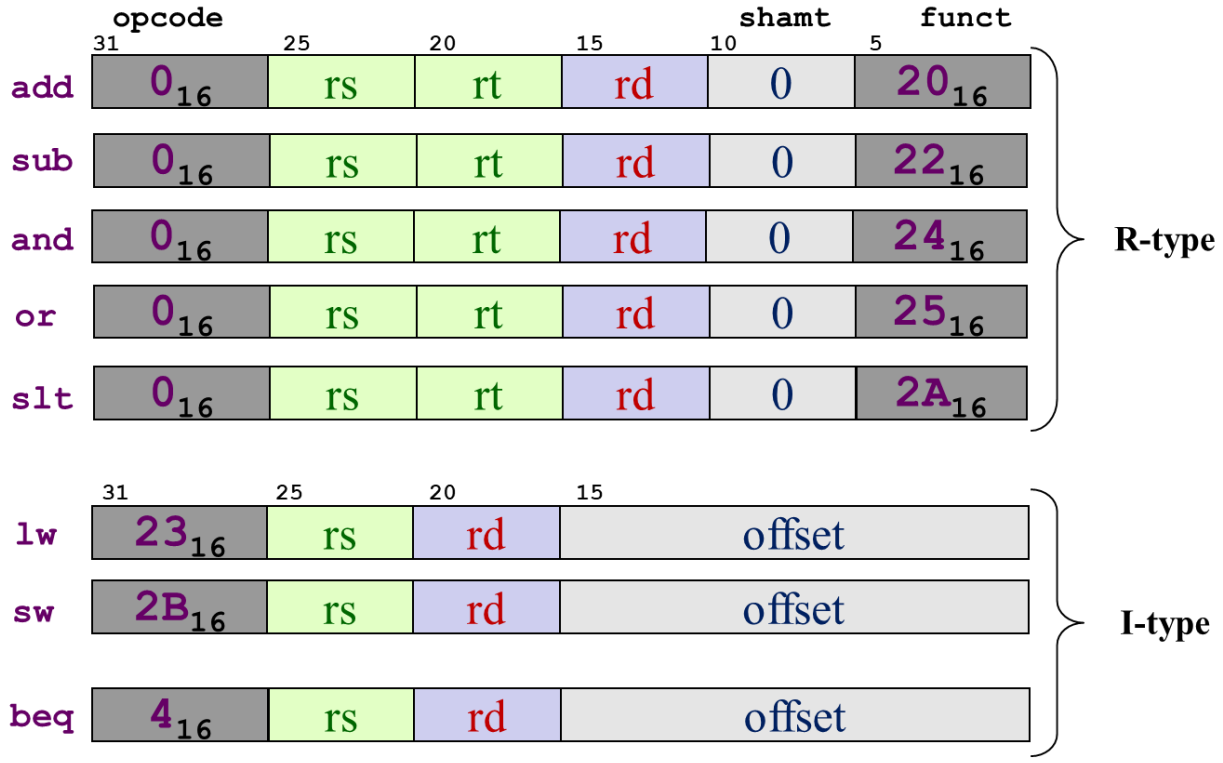
8.3 The Control Units



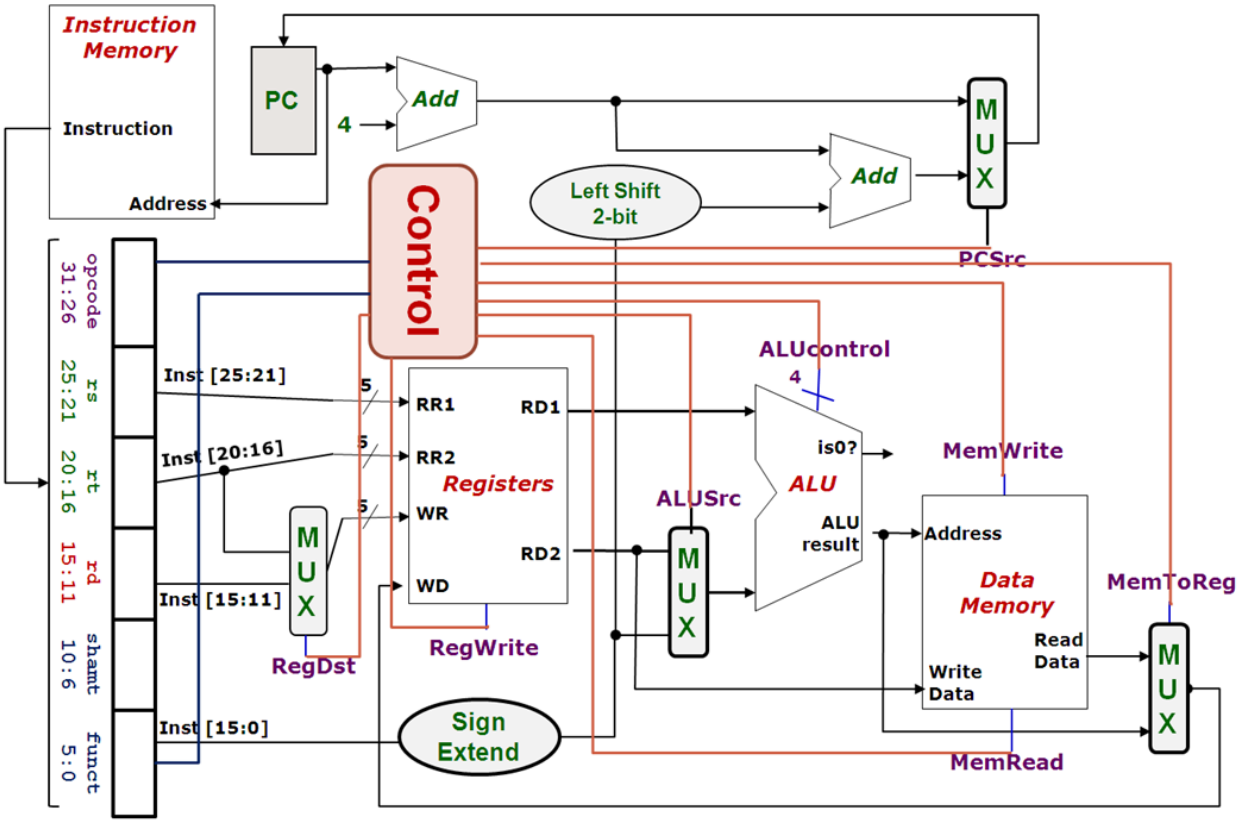
8.3.1 Implement the Control Unit

- 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



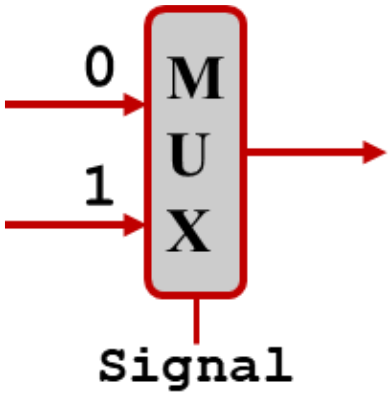
8.4 Control Signals



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

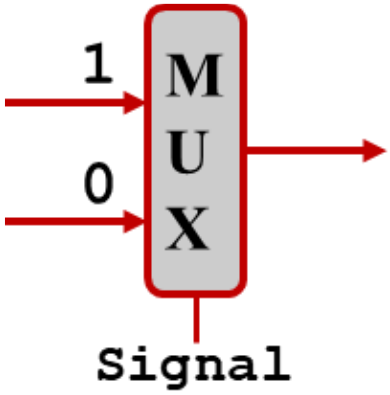
> 决定是否执行内存写入操作

- 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

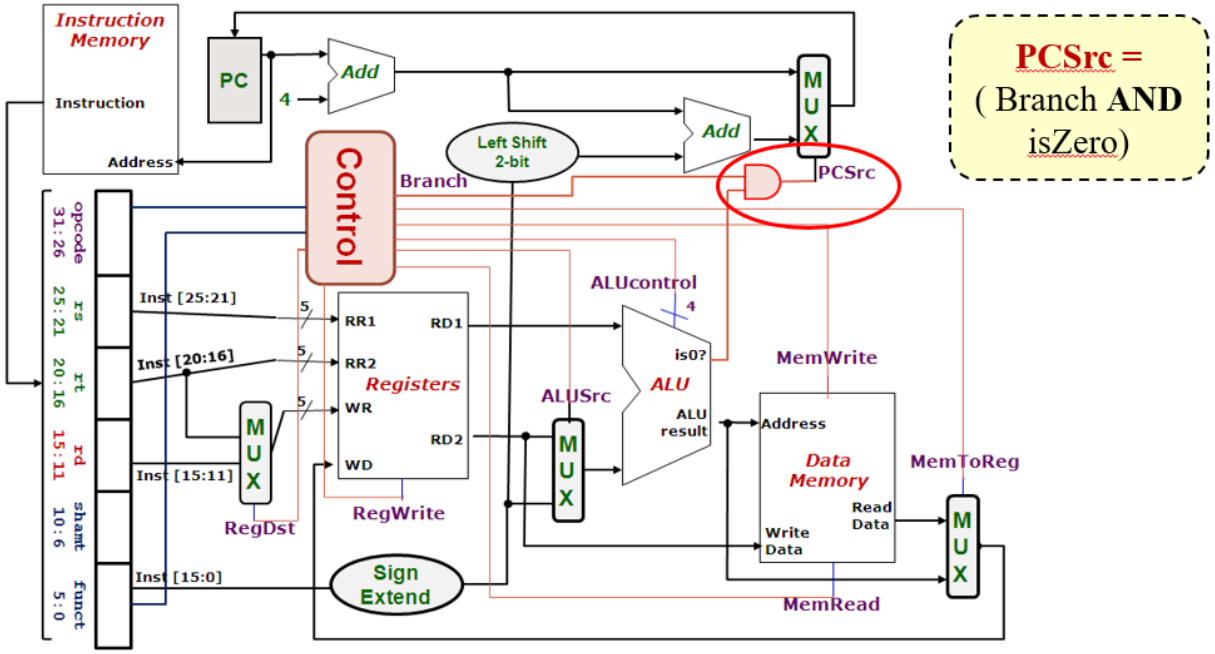
- True (1): Register write data = Memory read data
- False (0): Register write data = ALU result
- Important: The input of MUX is swapped in this case



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

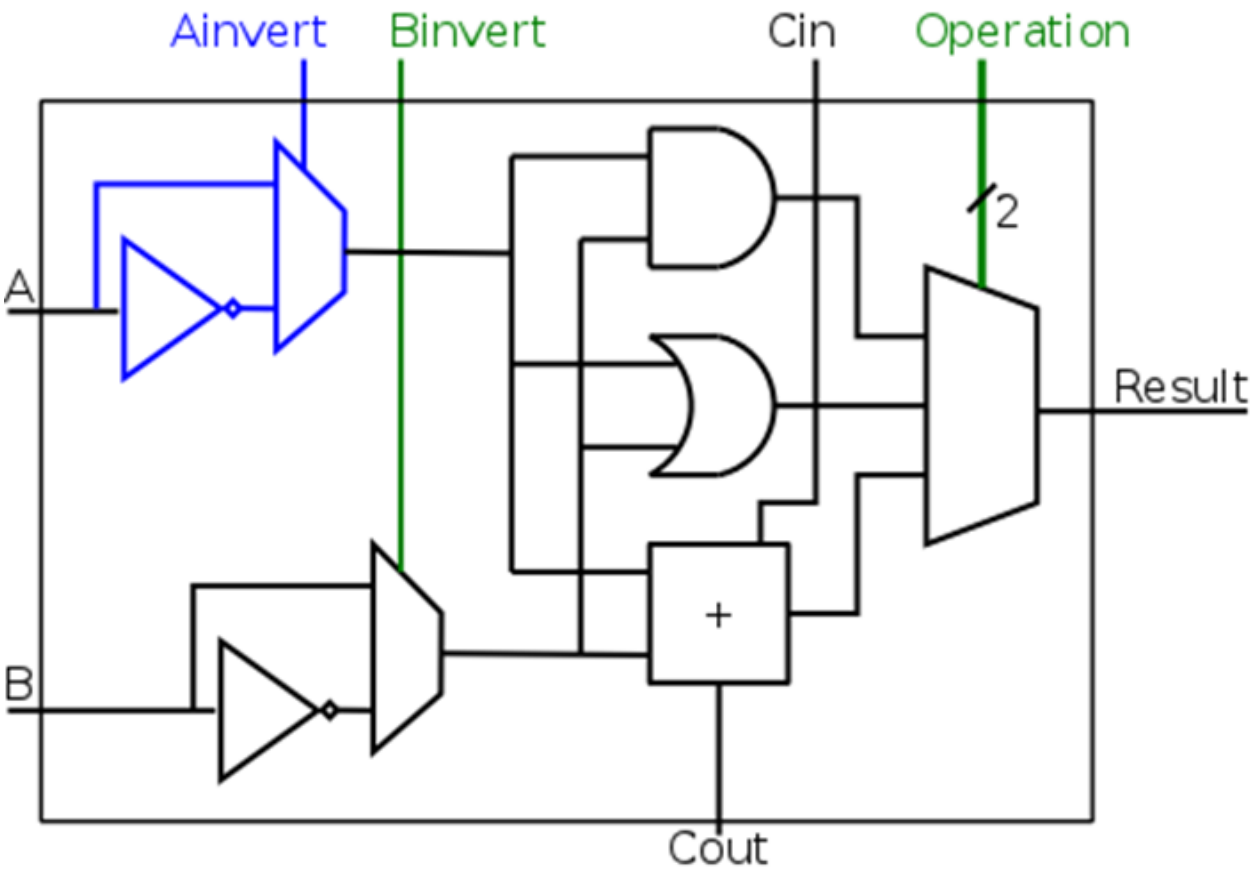
8.5 ALU Control Signal

- 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）。

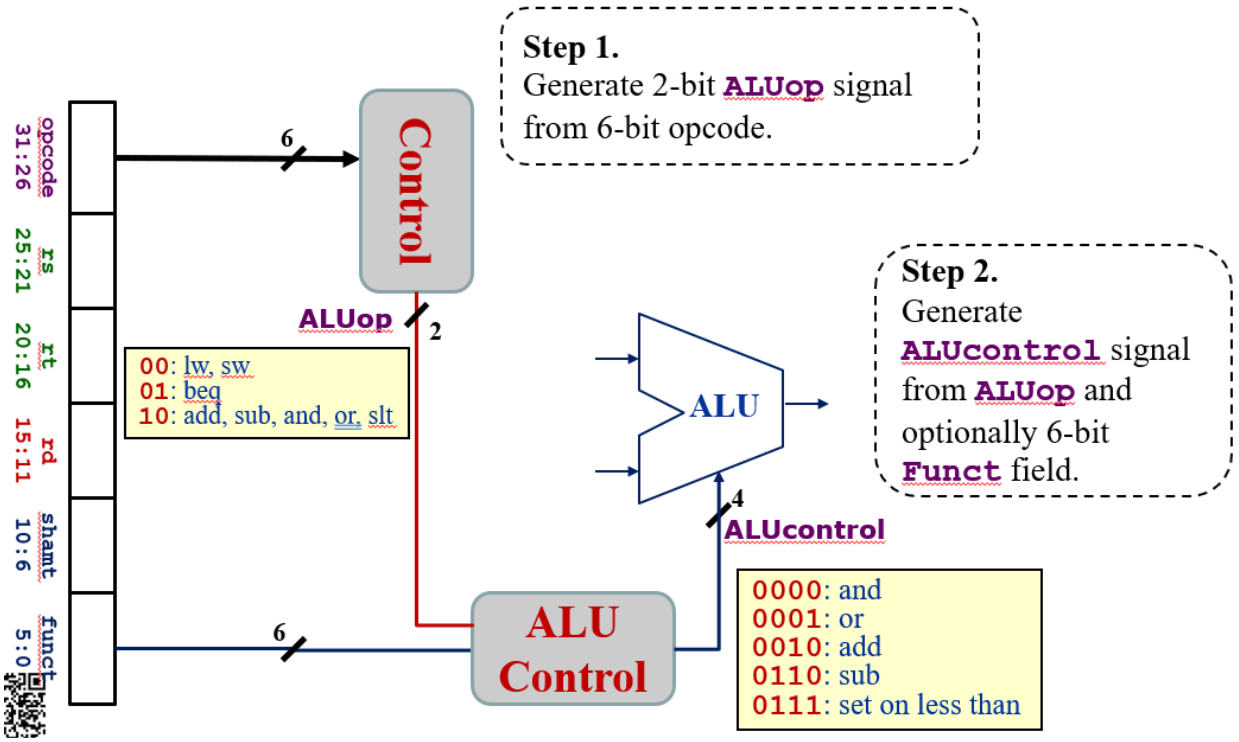
为什么需要 ALUop ：

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

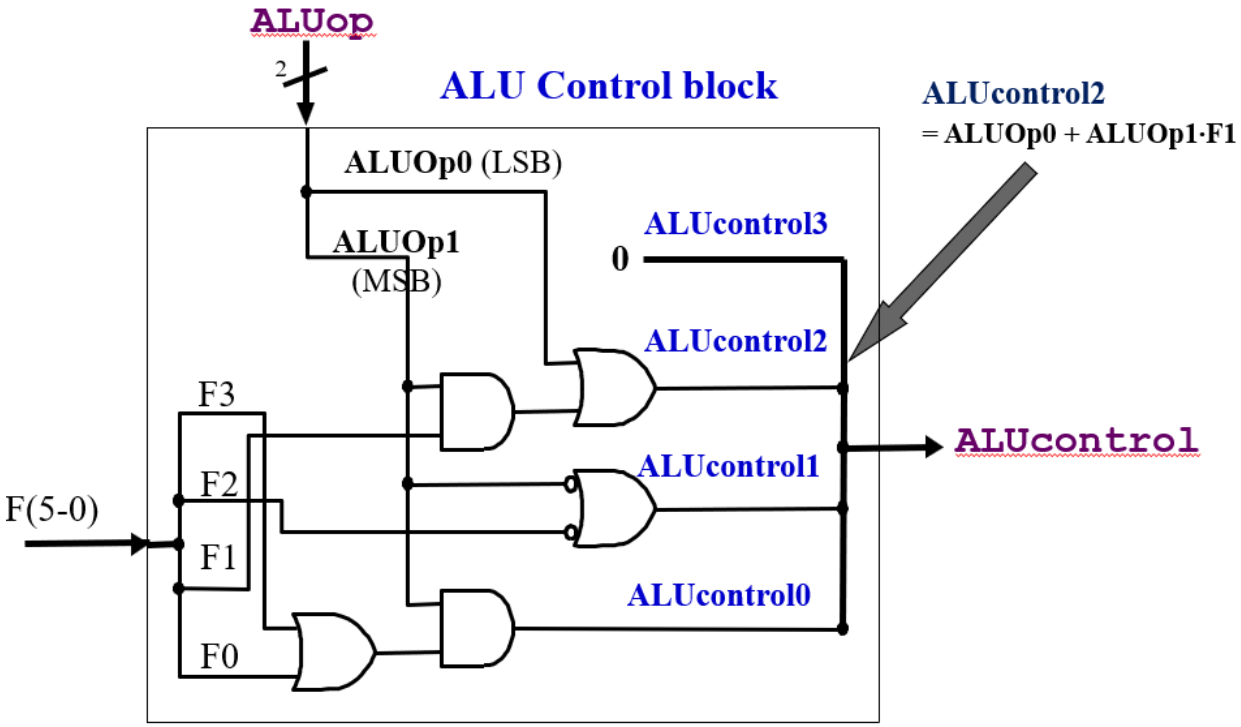
Generation of 2-bit `ALUop` signal will be discussed later

8.5.6 Design of ALU Control Unit

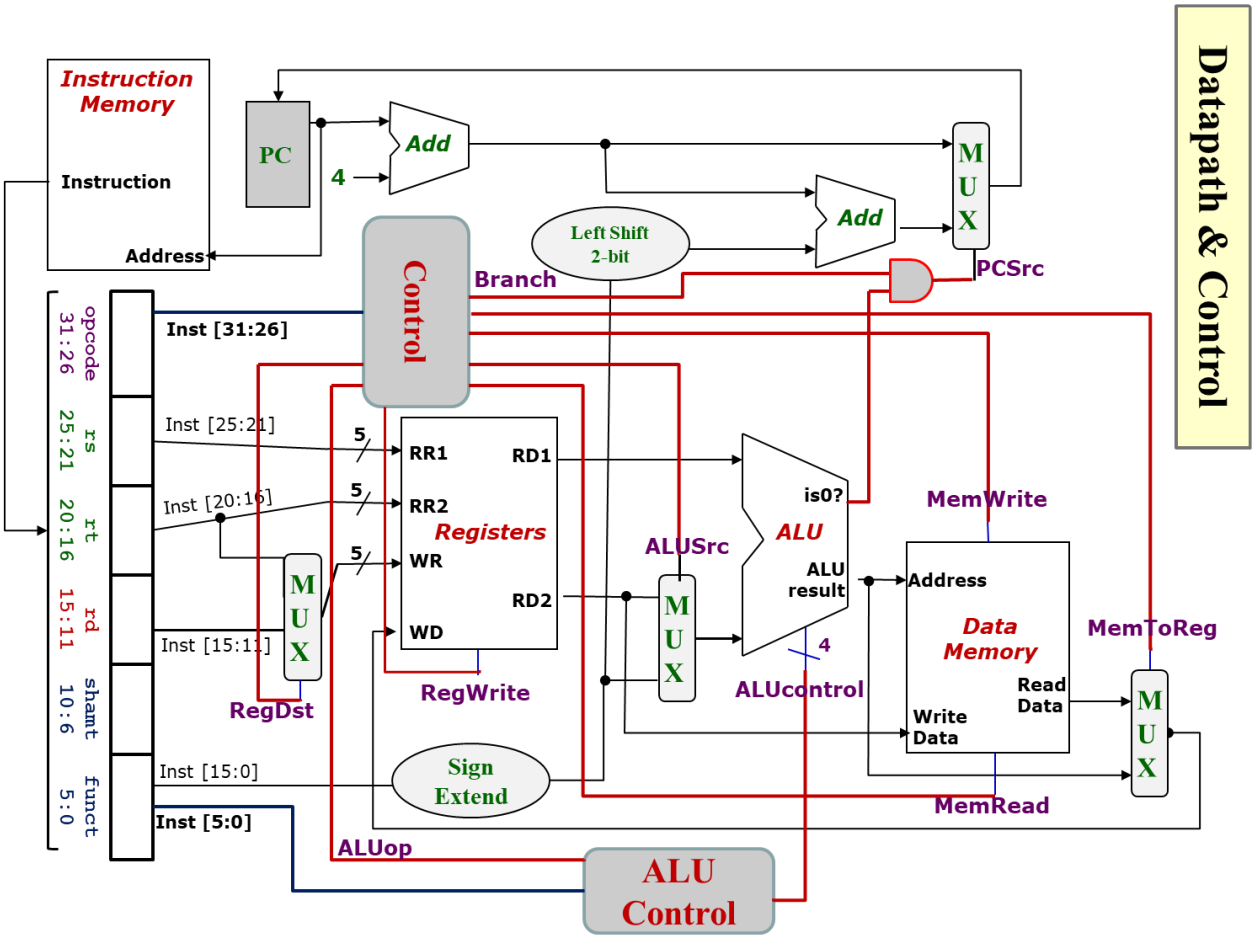
- Input: 6-bit `funct` field and 2-bit `ALUop`
- Output: 4-bit `ALUcontrol`

	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



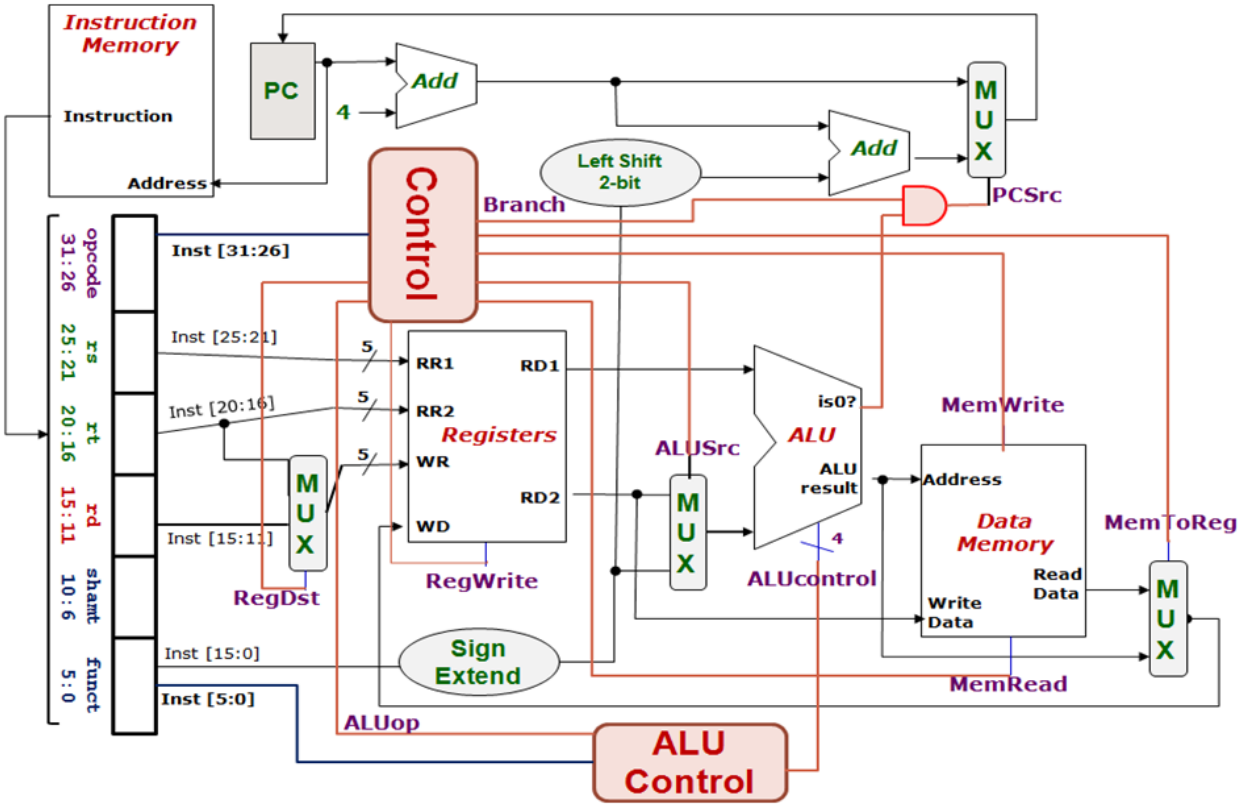
8.5.7 Summary



- 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

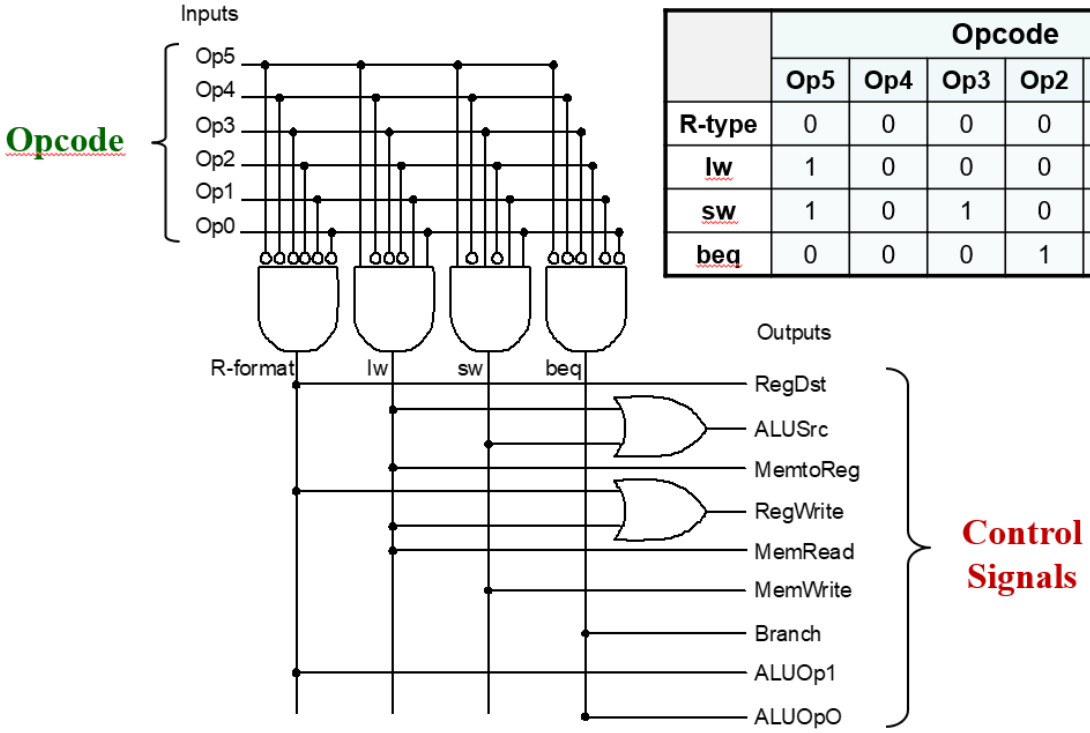
	RegDst	ALUSrc	MemTo Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop	
								op1	op0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1



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



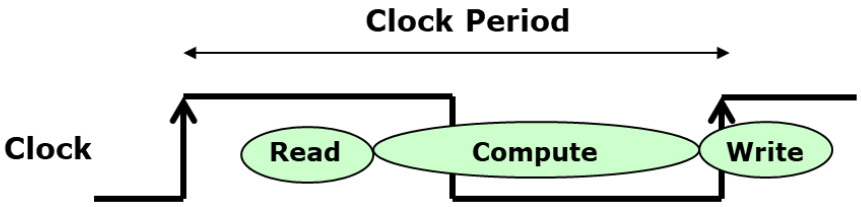
	Opcode					
	Op5	Op4	Op3	Op2	Op1	Op0
R-type	0	0	0	0	0	0
lw	1	0	0	0	1	1
sw	1	0	1	0	1	1
beq	0	0	0	1	0	0

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

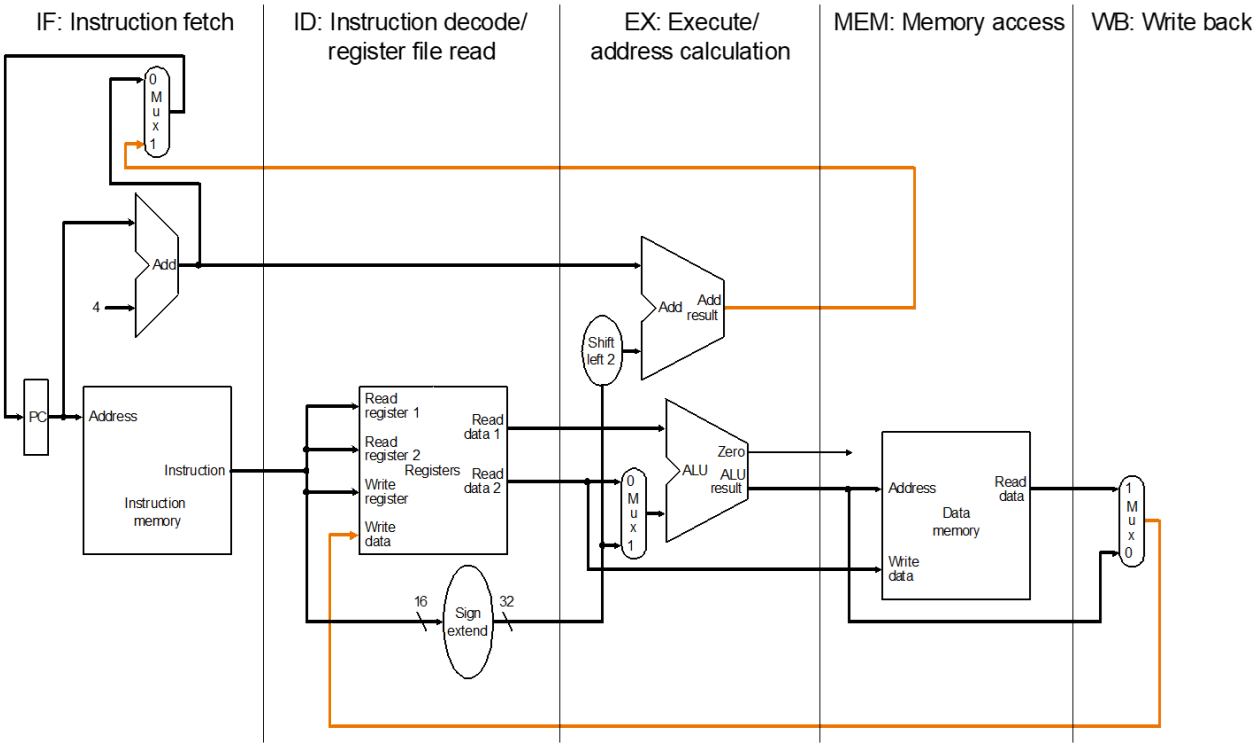
Idea:

- Each execution stage takes 1 clock cycle

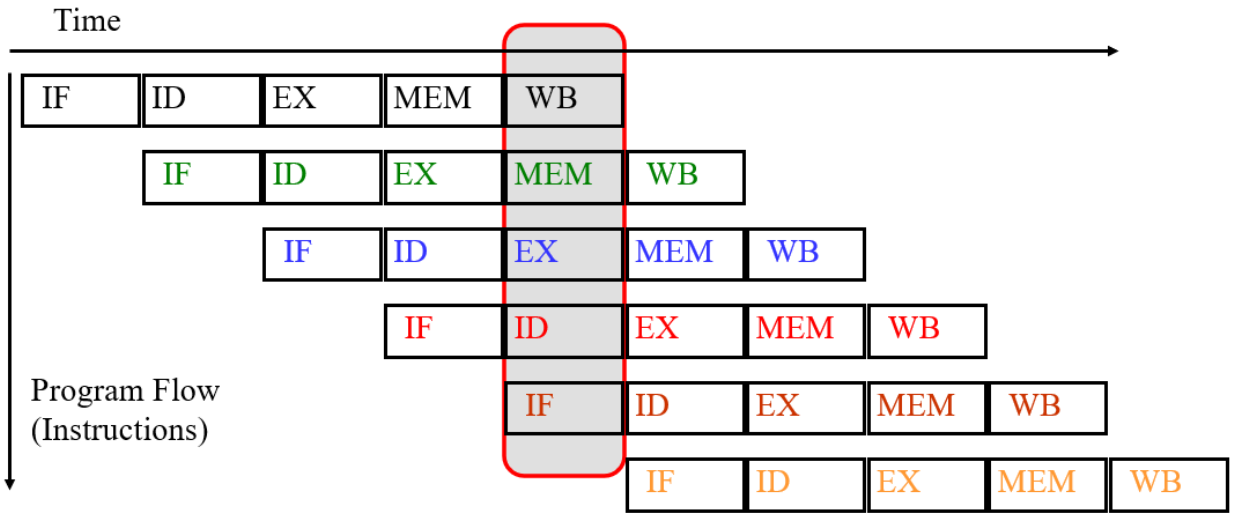
- General flow of data is from one stage to the next

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：

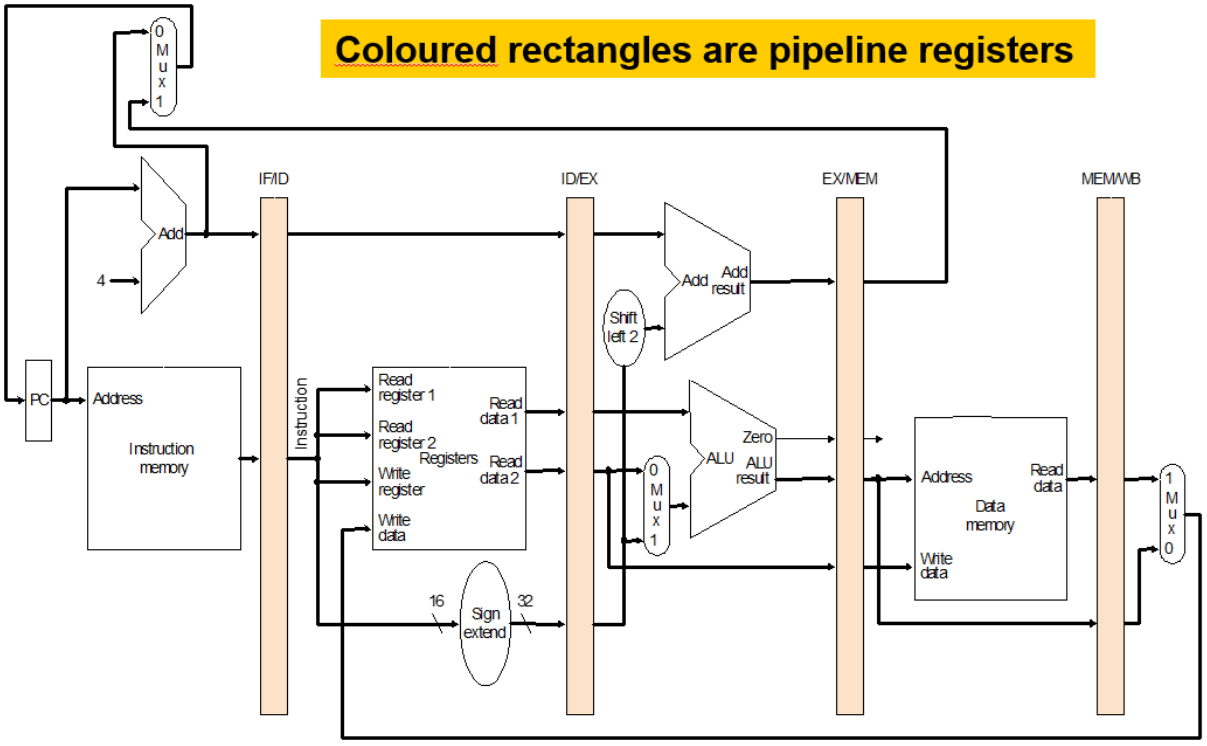
- **程序计数器（PC）**：它存储的是下一条要执行的指令的地址。
- **寄存器文件**：这是一组寄存器，用于存储在指令执行过程中需要的数据。
- **内存**：这是一个更大的数据存储区域，用于存储指令以及指令操作的数据。

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。

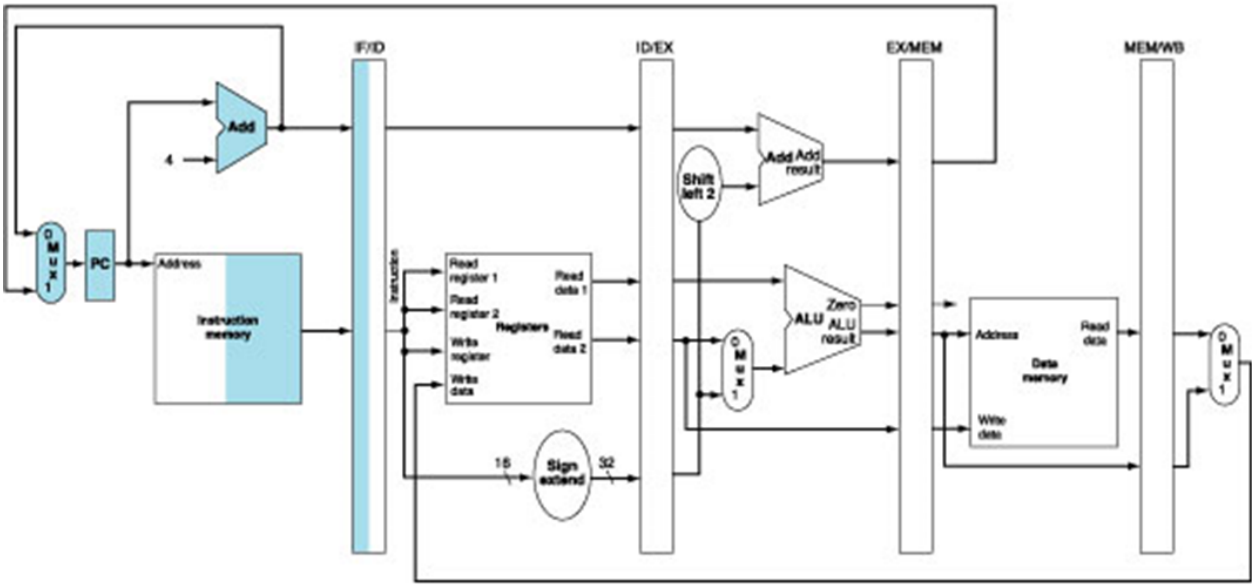
流水线寄存器：

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

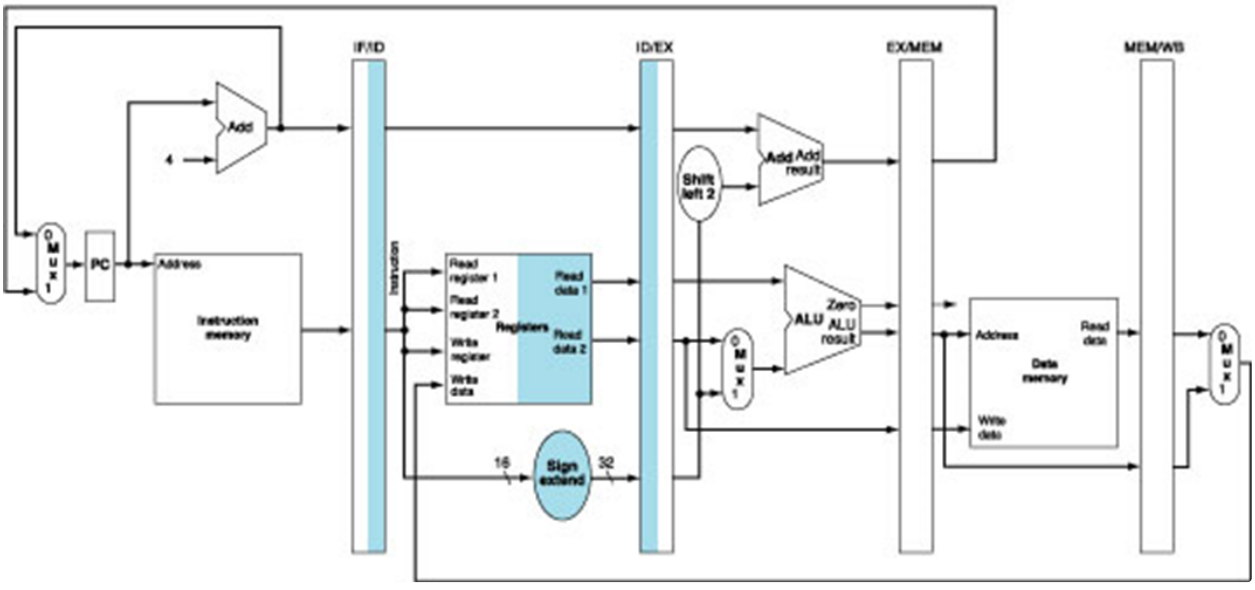


At the end of a cycle, IF/ID receives (stores):

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

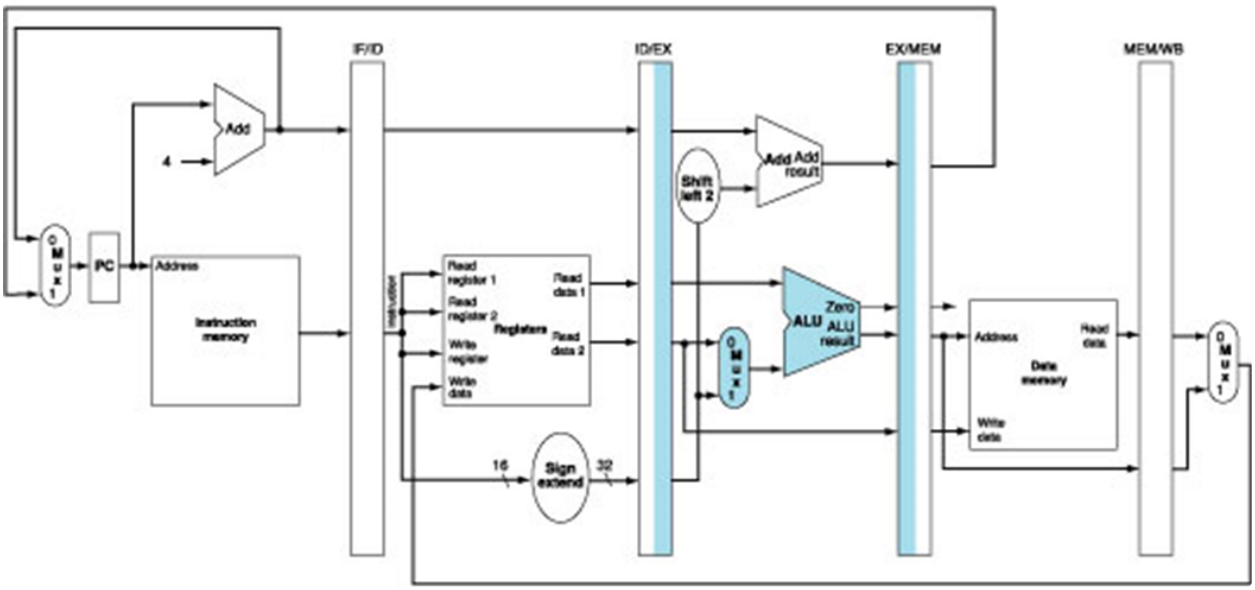
PC+4 also connected to one of the MUX's inputs

9.3.2 ID Stage



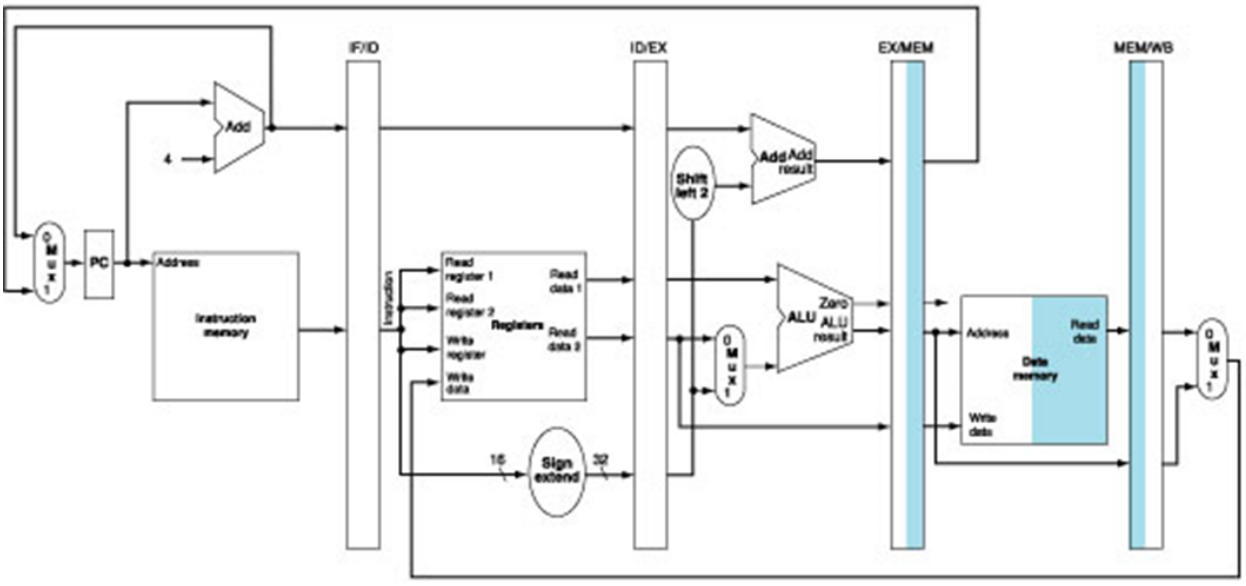
At the beginning of a cycle IF/ID register supplies:	At the end of a cycle ID/EX receives:
❖ Register numbers for reading two registers ❖ 16-bit offset to be sign-extended to 32-bit	❖ Data values read from register file ❖ 32-bit immediate value ❖ **PC + 4**

9.3.3 EX Stage



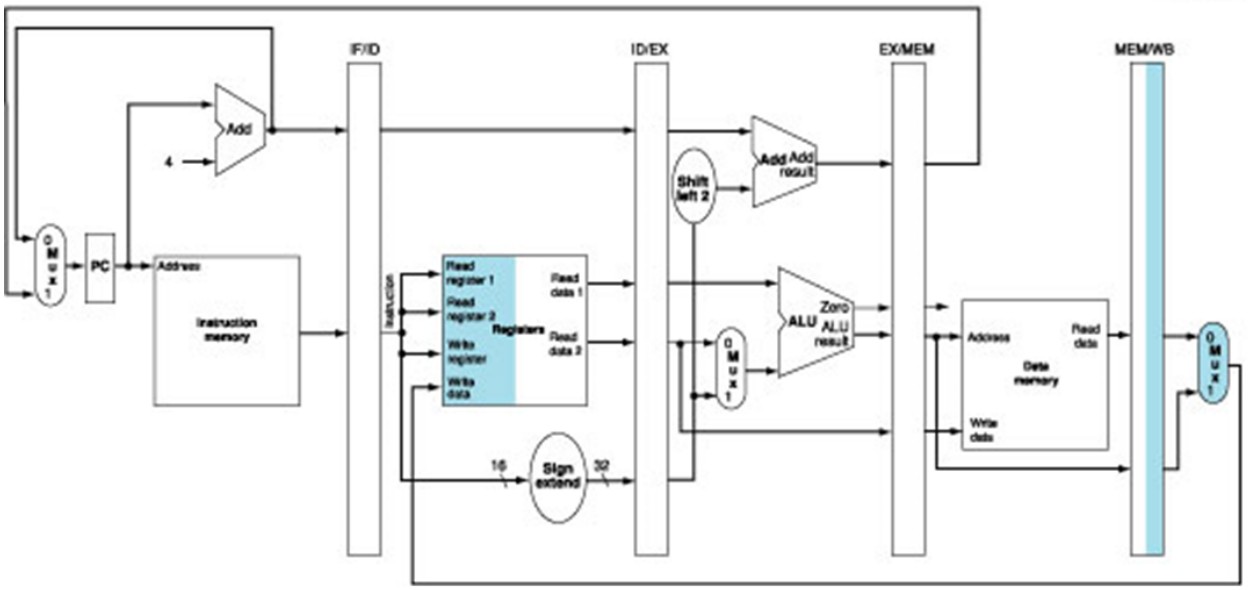
At the beginning of a cycle ID/EX register supplies:	At the end of a cycle EX/MEM receives:
❖ Data values read from register file ❖ 32-bit immediate value ❖ **PC + 4**	❖ (PC + 4) + (Immediate x 4) ❖ ALU result ❖ **isZero?** signal ❖ Data Read 2 from register file

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:

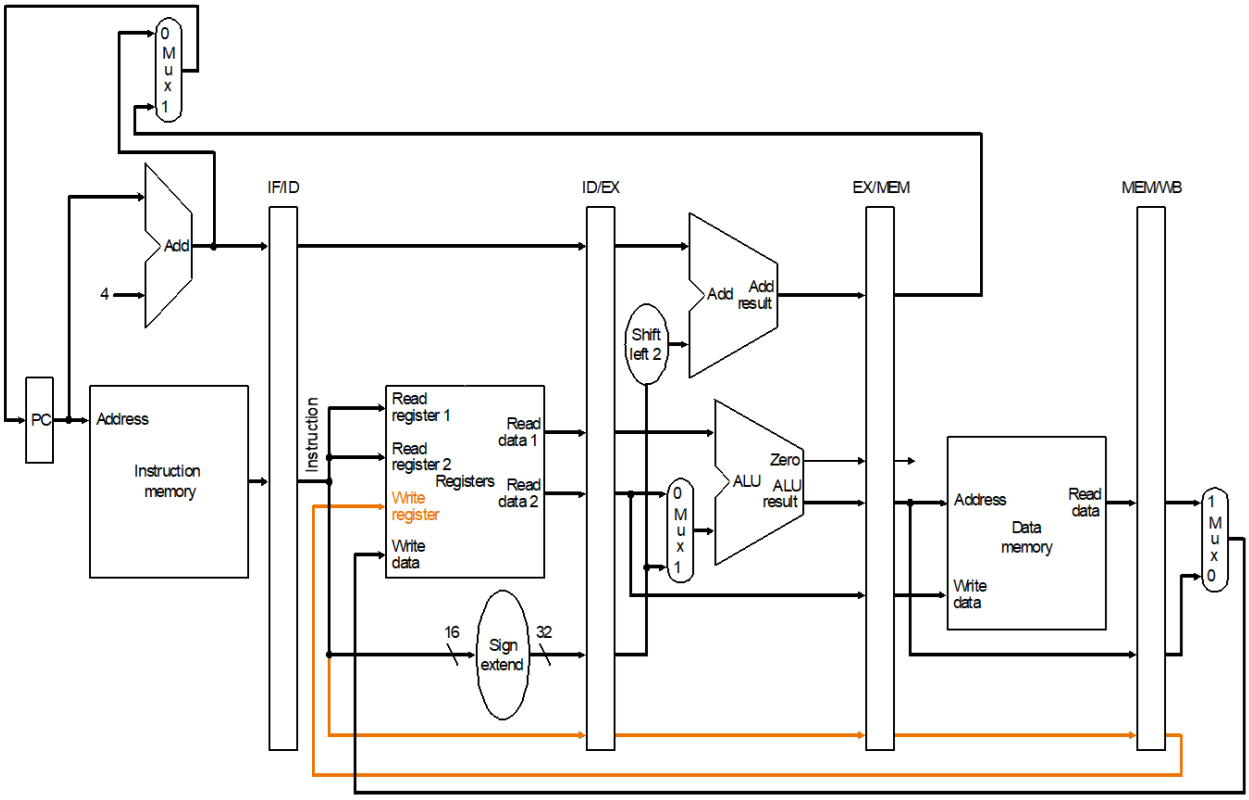
- Pass “Write register” number from `ID/EX` through `EX/MEM` to `MEM/WB` pipeline register for use in `WB` stage
- i.e. let the “Write register” number follows the instruction through the pipeline until it is needed in `WB` stage

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。

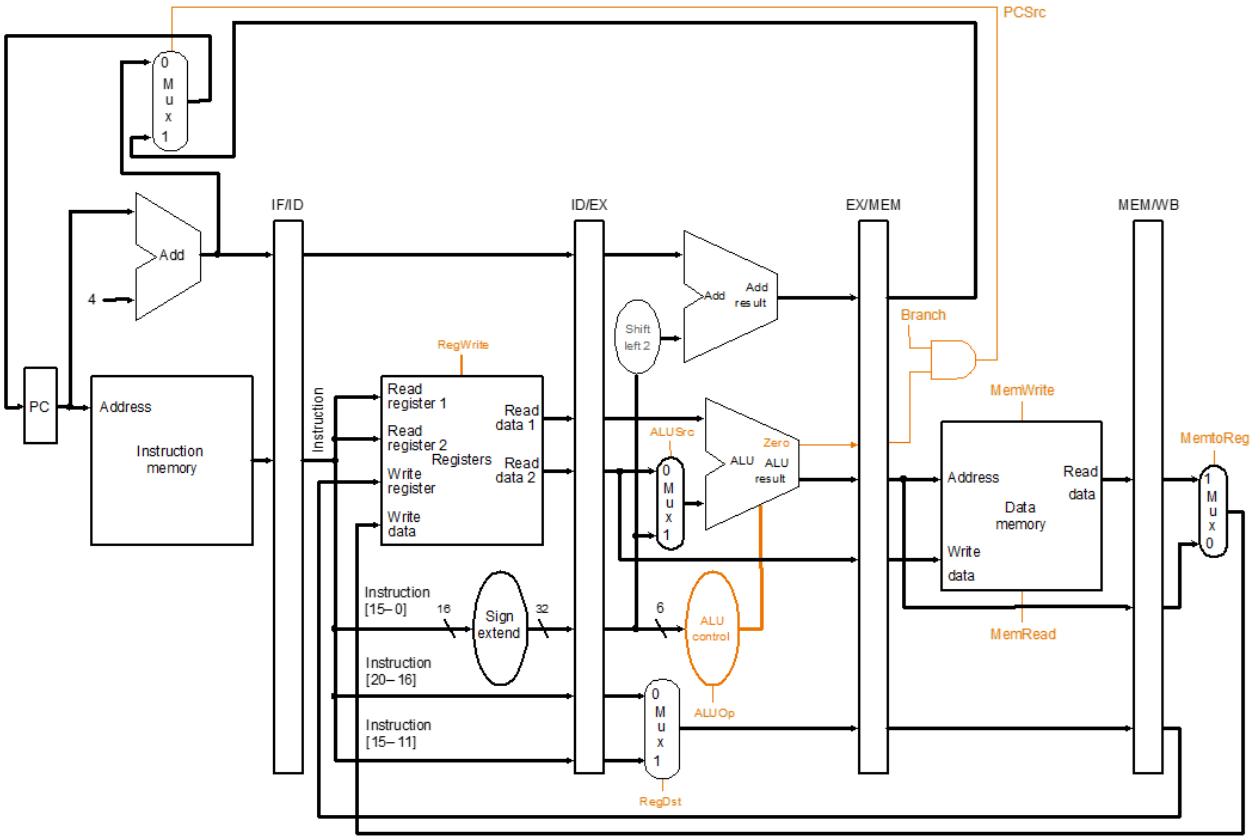
总的来说，这种更正确保了数据的准确性，并使流水线在处理每条指令时能够维持正确的状态。这是流水线设计中处理各种依赖关系的通用方法之一，确保数据和控制信息能够在需要时可用，从而避免错误和性能下降。



9.4 Pipeline Control

Main Idea

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



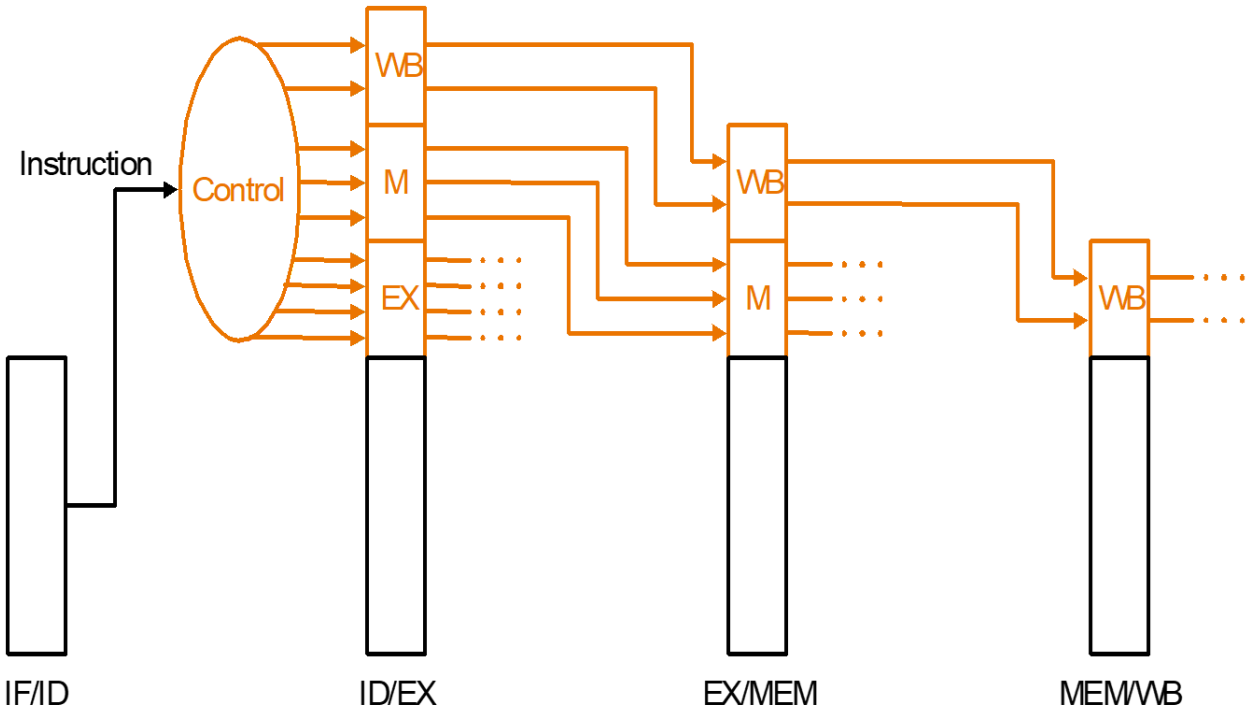
9.4.1 Grouping

Group control signals according to pipeline stage

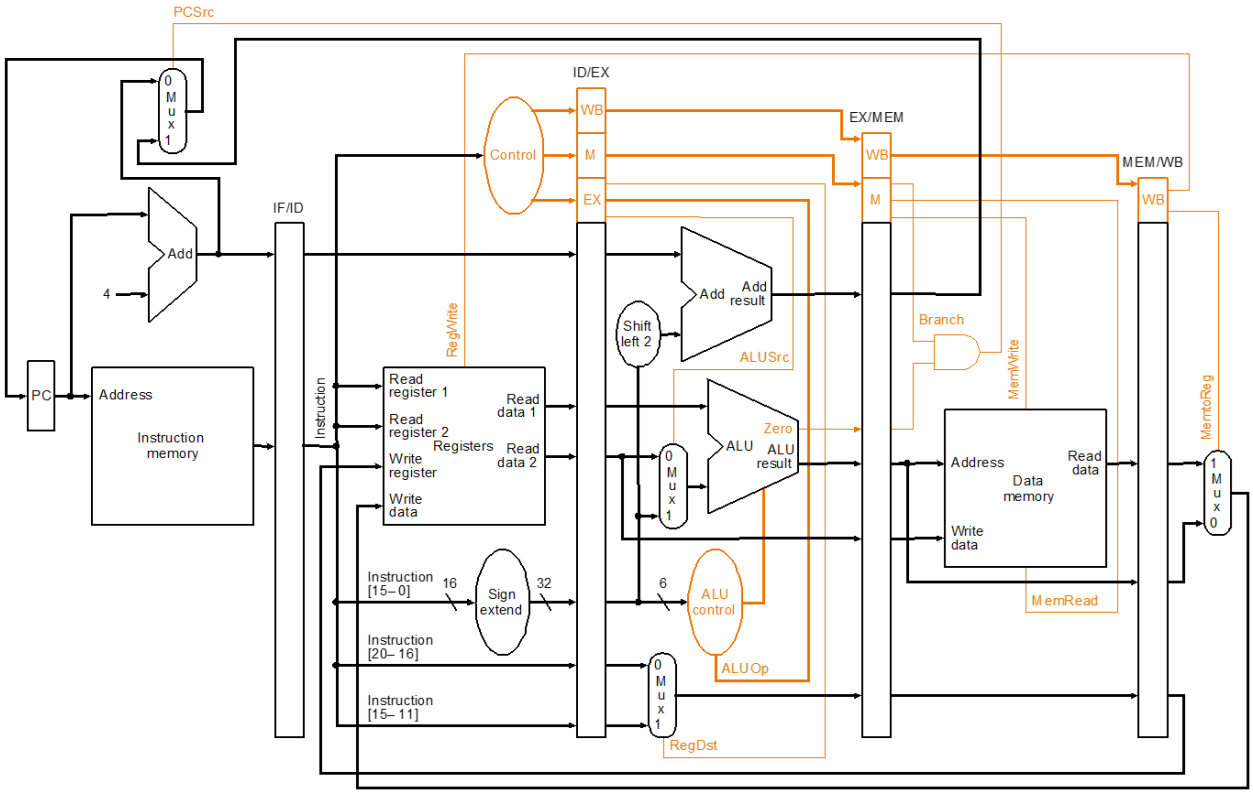
	RegDst	ALUSrc	MemTo Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop	
								op1	op0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

	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

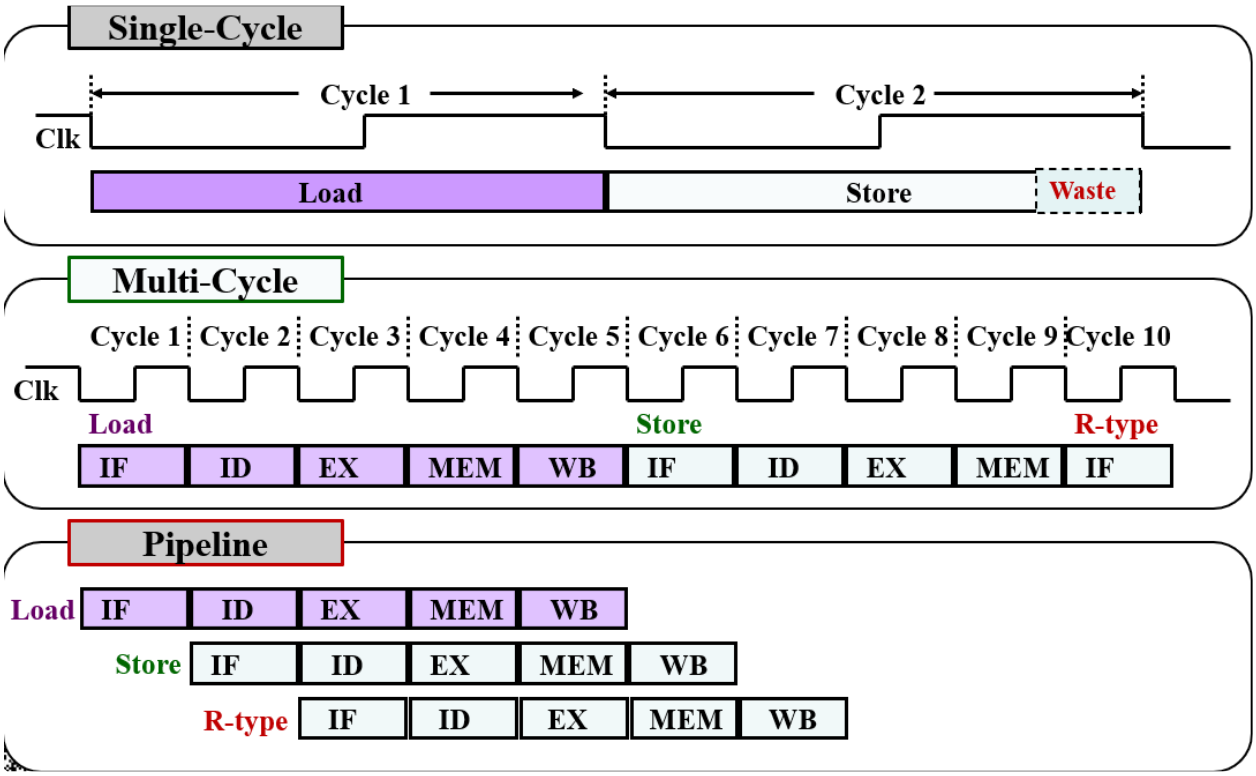


9.4.3 Datapath and Control



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:

- $T_{seq} =$ Cycles $\times$ 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 -> $T_d = 0$
- 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 $Speedup_{pipeline}$。

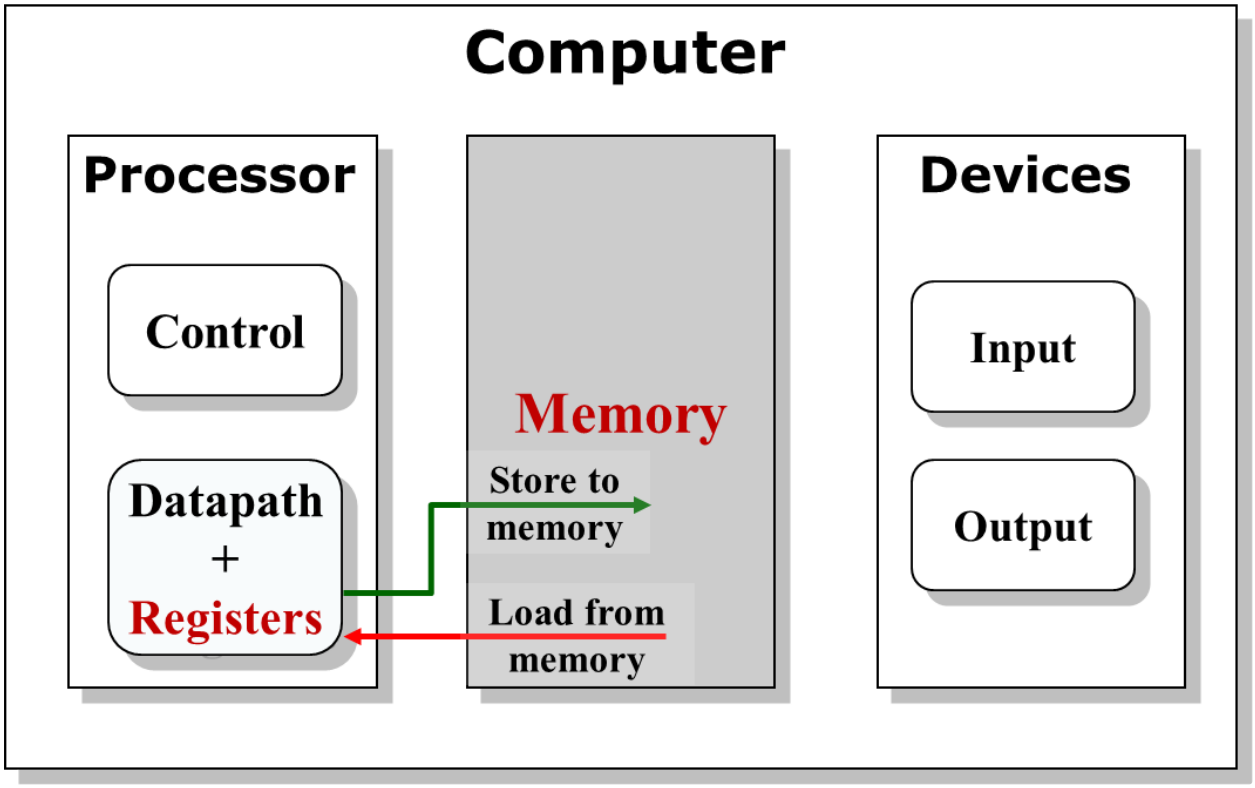
1. **顺序执行时间** 是所有指令在非流水线设置中执行所需的总时间，计算为指令数 I 乘以每条指令的执行时间（由于每个阶段都花费 T_k 时间，总时间为 $N \times T_k$）。
2. **流水线执行时间** 是在流水线设置中执行所有指令所需的时间。重要的是要注意，第一条指令完成之后，每个额外的周期都会完成一条新指令。因此，总时间包括了初始的 N 个阶段和随后的 $I-1$ 个周期（每条指令一个），再加上流水线的开销时间 T_d，但在我们的理想假设中，$T_d = 0$。
3. **加速比** 是顺序时间与流水线时间的比率。在这种情况下，因为每个阶段花费相同的时间并且没有额外的流水线开销，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

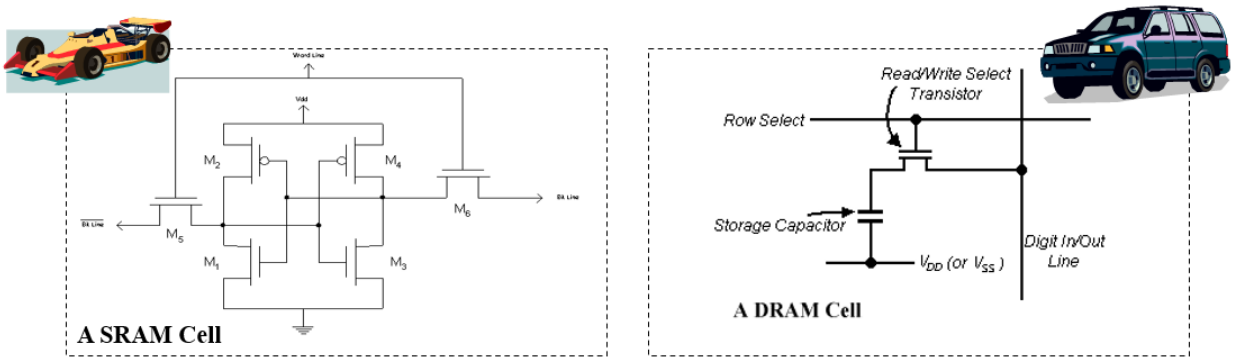
DRAM = DDR SDRAM = Double Data Rate Synchronous Dynamic RAM

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

Generations:

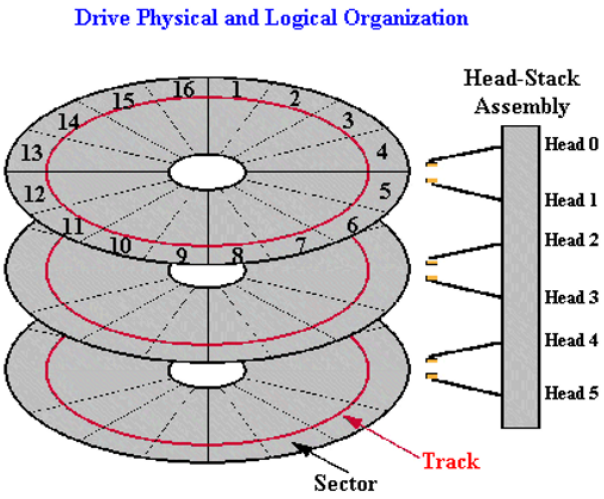
1. DDR (`MemClkFreq x 2 (double rate) x 8 words`)
2. DDR2 (`MemClkFreq x 2 (double rate) x 2 x 8 words`)
3. DDR3 (`MemClkFreq x 4 (double rate) x 2 x 8 words`)
4. DDR4 (Lower power consumption, higher bandwidth)

SRAM



SRAM	DRAM
6 transistors per memory cell	1 transistor per memory cell
→ **Low density**	→ **High density**
Fast access latency of 0.5 – 5 ns	**Slow access** latency of 50-70ns
More costly	Less costly
Uses flip-flops	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

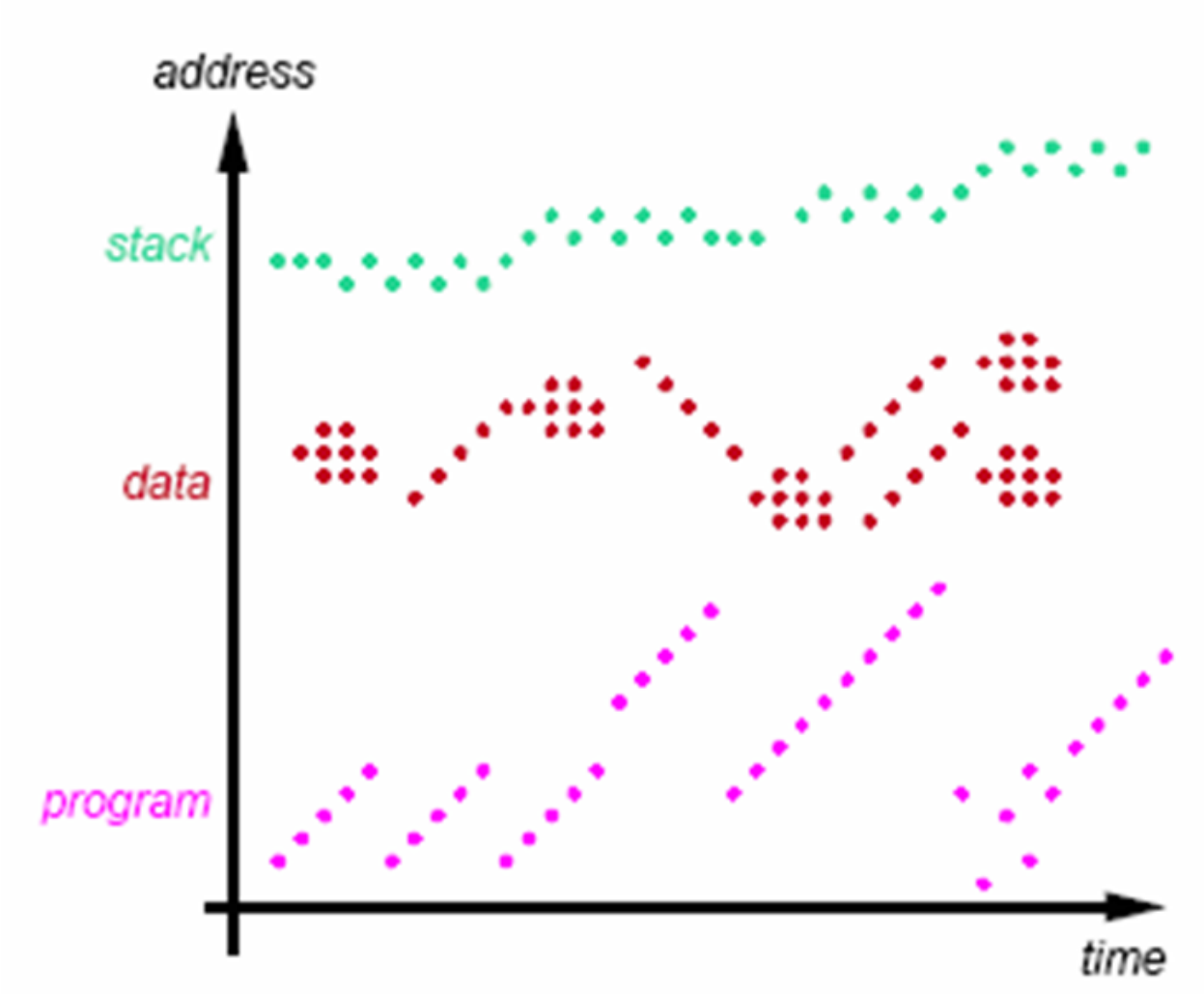
10.2.1 Types of locality

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。

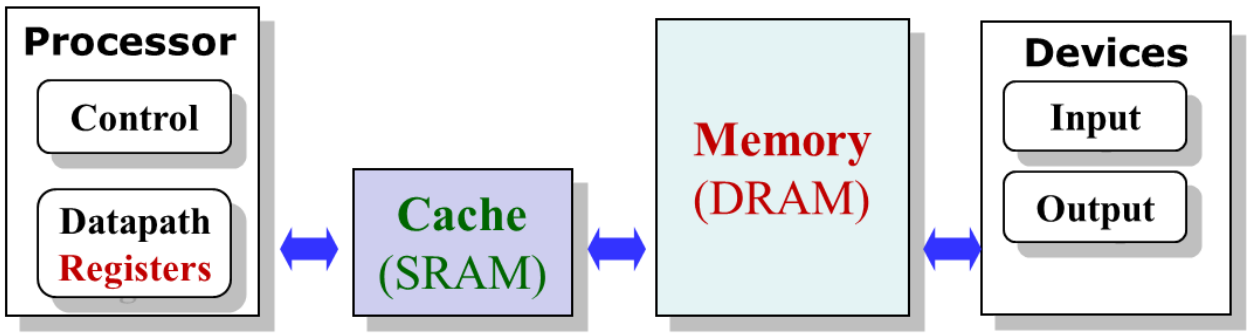
执行阶段和工作集

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



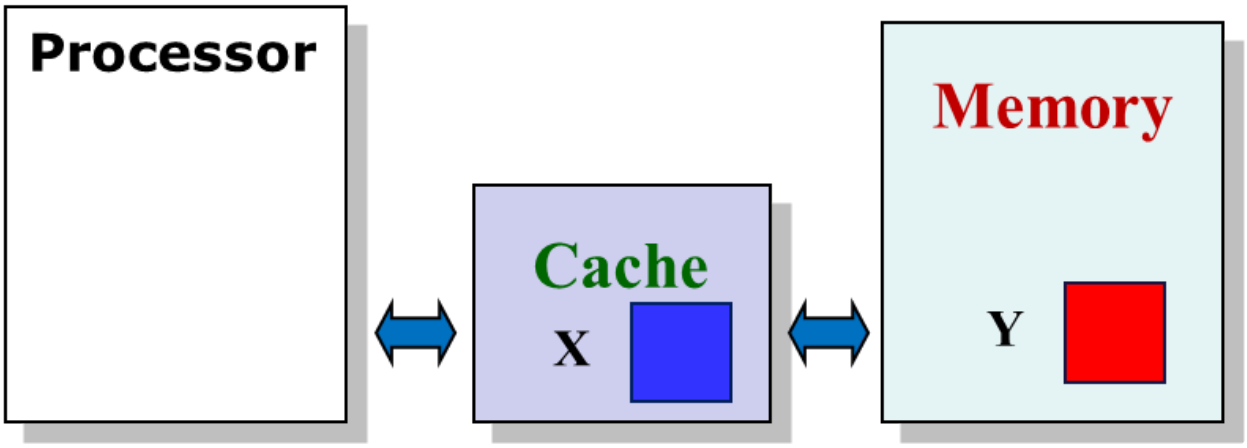
How to make SLOW main memory appear faster?

- 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) x (10 + 0.8)
      = 0.8h + 10.8 - 10.8h

    10h = 9.8 -> 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

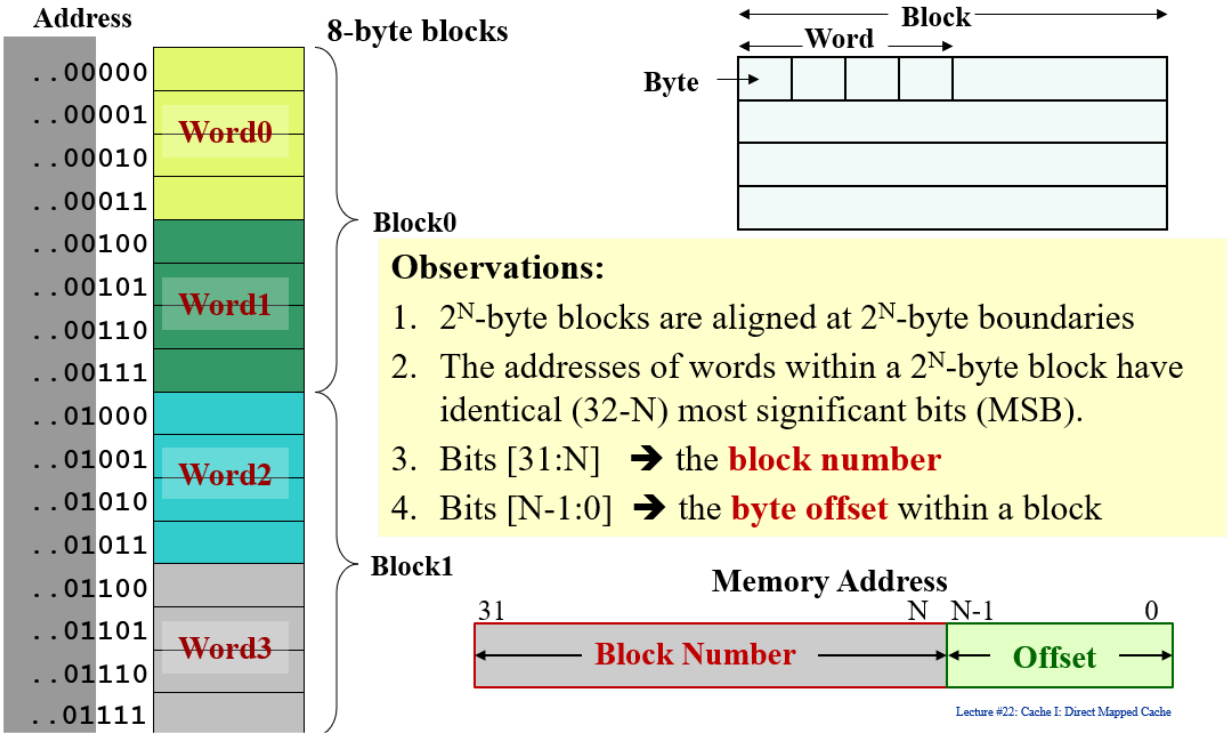
Block size is typically one or more words

- 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。

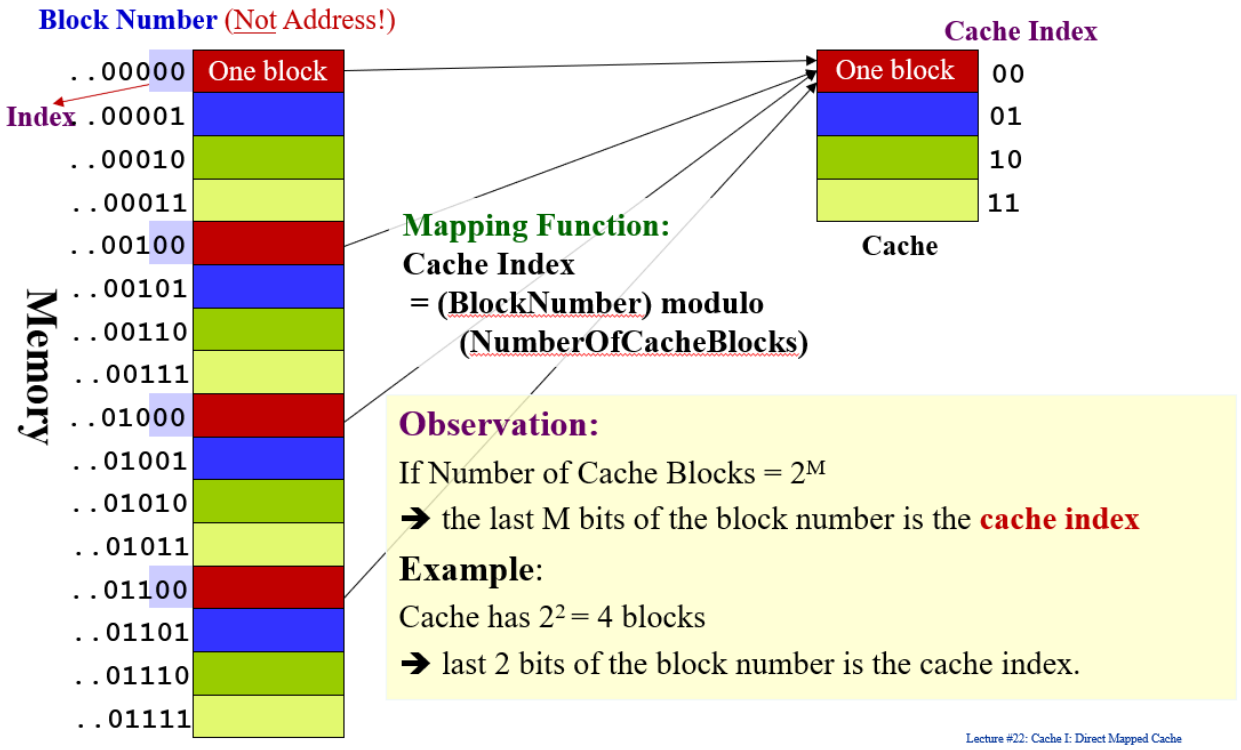
为了决定一个内存块应该存储在缓存的哪个位置，以及如何在缓存中找到这些块，系统会使用一种映射策略。在直接映射缓存中，使用了内存地址的一部分来确定一个特定的缓存线。

缓存索引的作用：

内存地址通常包含以下几个部分：

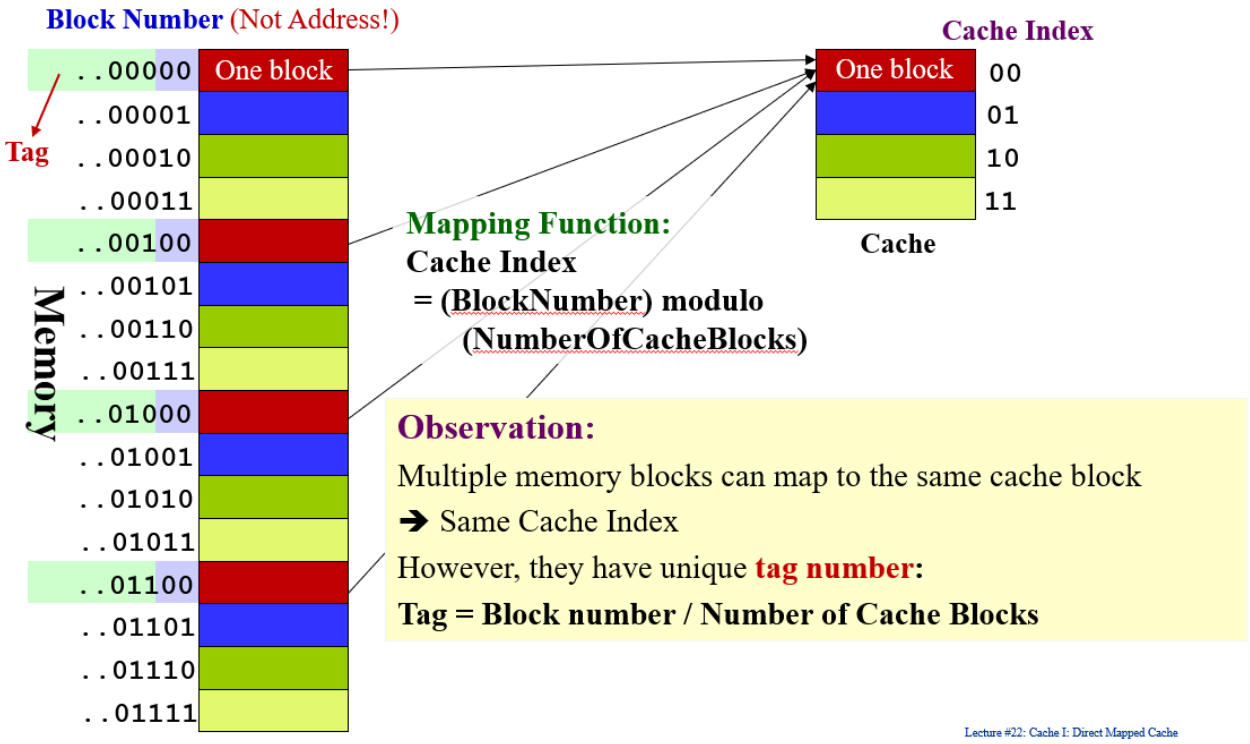
1. **标记（Tag）**：用于唯一标识一个内存块。当缓存检查某个特定缓存线时，它会比较地址标记和缓存线中存储的标记，以确定所需数据是否在该缓存线中。
2. **索引（Index）**：这是直接决定内存块在缓存中位置的部分。系统通过内存地址的索引字段来选择应该使用哪个缓存线。换句话说，索引用于“直接映射”内存块到特定的缓存线。
3. **块内偏移（Block Offset）**：当存储块大小大于一个字（word）时，块内偏移用于在一个块内部定位特定的字。

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)

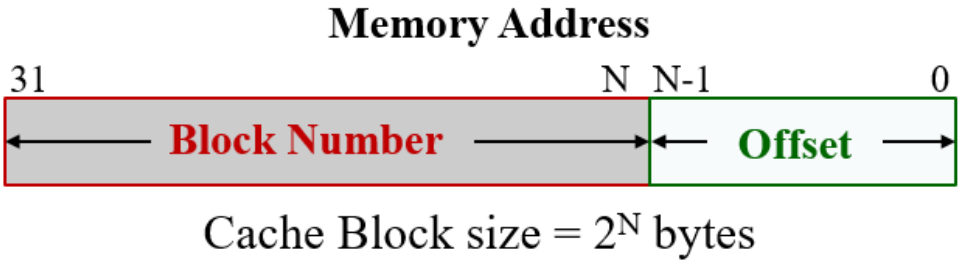
定义与功能:

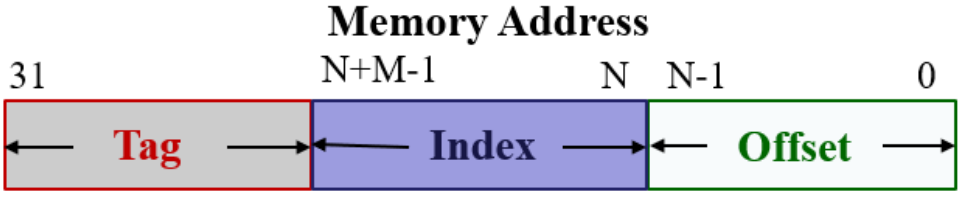
- "标记"是内存地址的一部分，用于在缓存中唯一标识一个数据块。由于缓存容量比整个内存小，因此不可能将所有内存块同时加载到缓存中。标记用于区分不同的内存块，即使它们可能映射到缓存的同一索引位置。
- 当处理器试图访问缓存时，系统会检查所选缓存行中存储的标记与当前请求的内存地址的标记部分是否匹配。如果这两个标记相同，就会发生“缓存命中”；否则，就会发生“缓存未命中”。

计算标记:

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

10.4.2 Mapping





Cache Block size = 2^N bytes

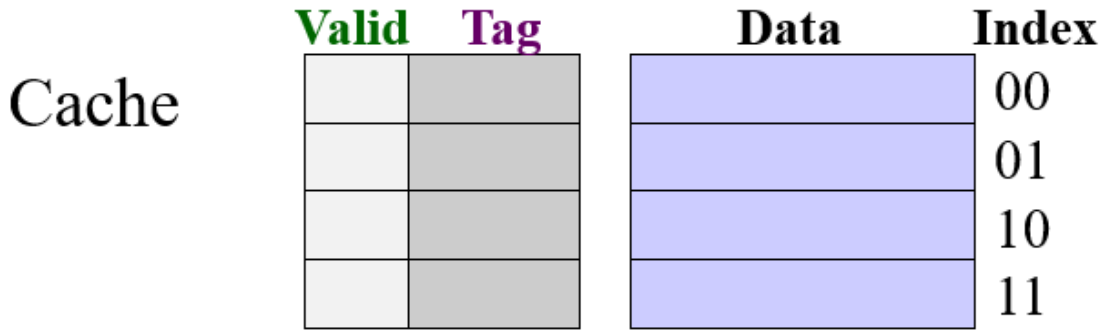
Number of cache blocks = 2^M

Offset = N bits

Index = M bits

Tag = 32 – (N + M) bits

10.4.3 Cache Structure



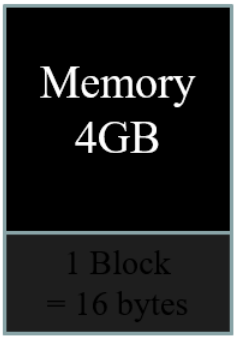
Along with a data block(line), cache also contains the following administrative information

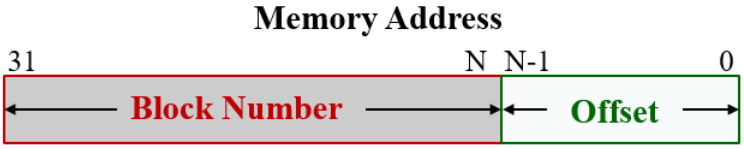
1. Tag of the memory block
2. Valid bit indicating whether the cache line contains valid data

When is there a cache hit?

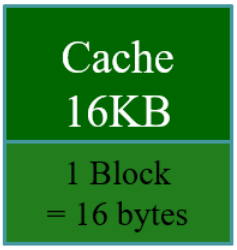
```
Valid[index] == True AND Tag[index] == Tag[memory address]
```

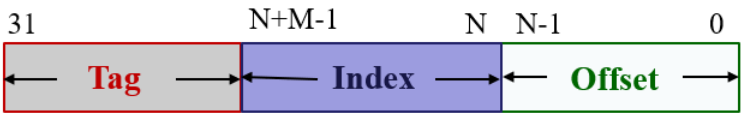
10.4.4 Example





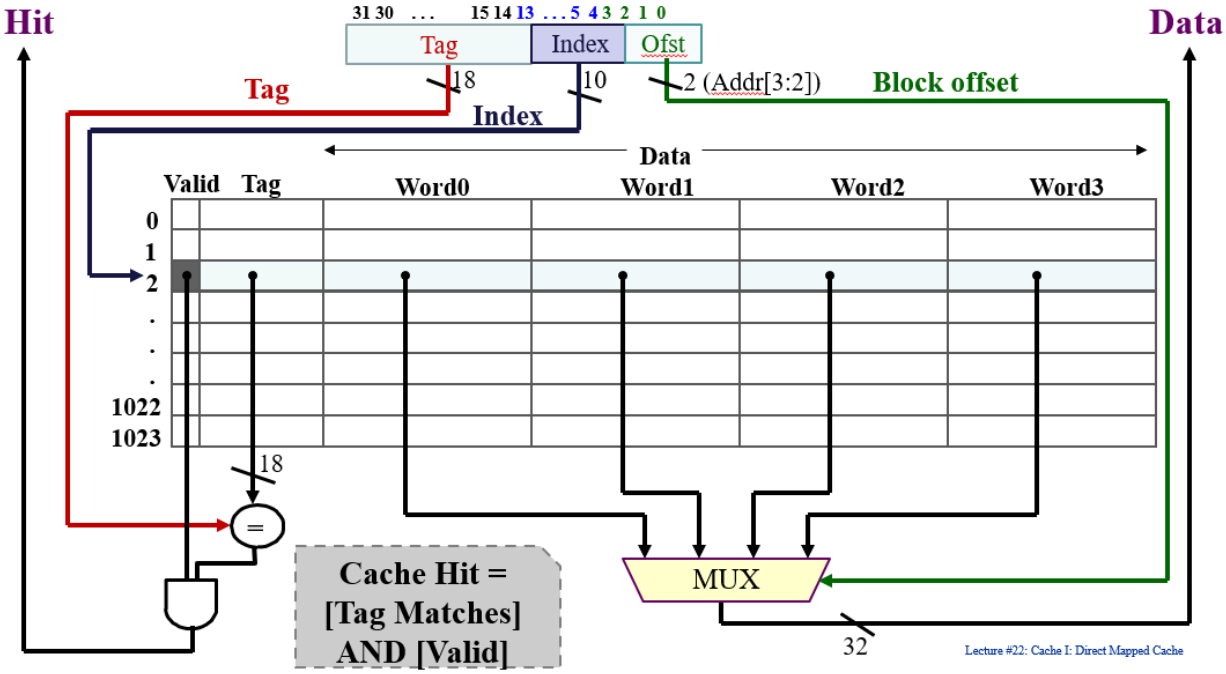
Offset, N = 4 bits
Block Number = 32 – 4 = **28 bits**
Check: Number of Blocks = 2^{28}





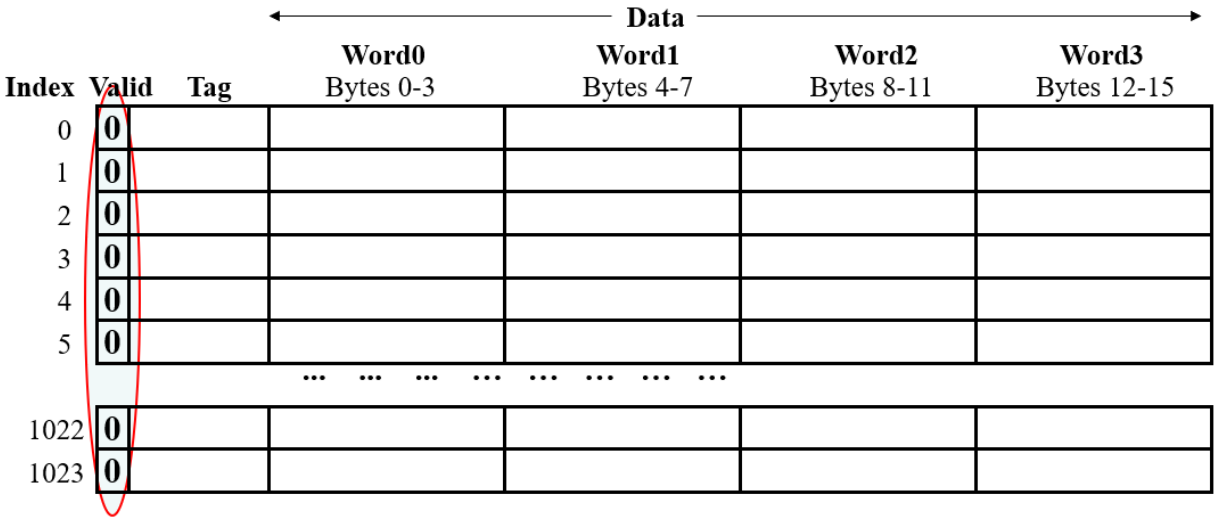
Number of Cache Blocks
= 16KB / 16bytes = 1024 = 2^{10}
Cache Index, M = 10bits
Cache Tag = 32 – 10 – 4 = **18 bits**

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



10.5 Reading Data

#1 Initial State



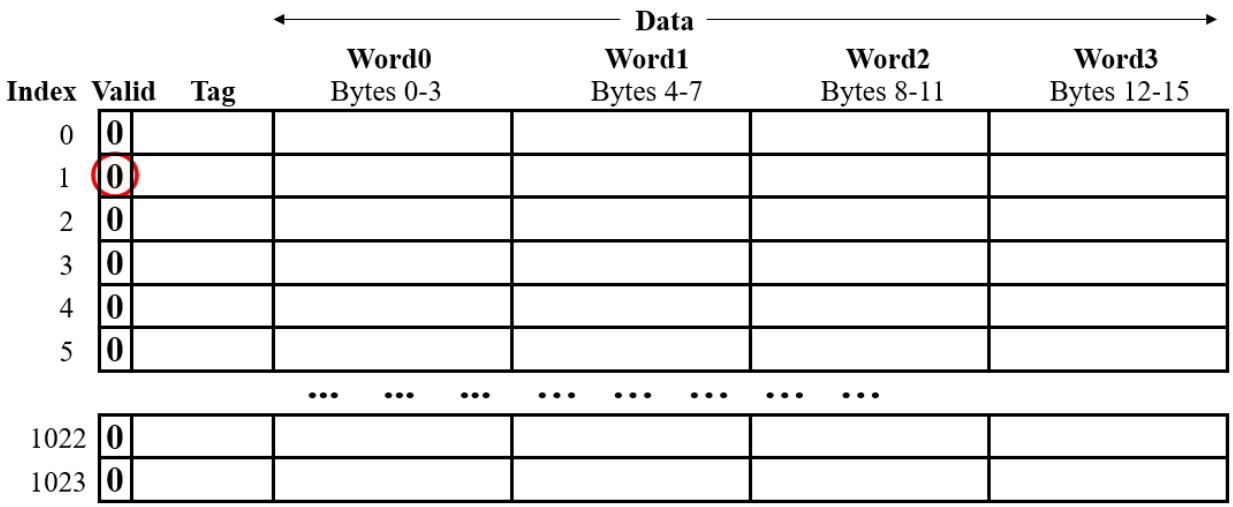
起始状态：

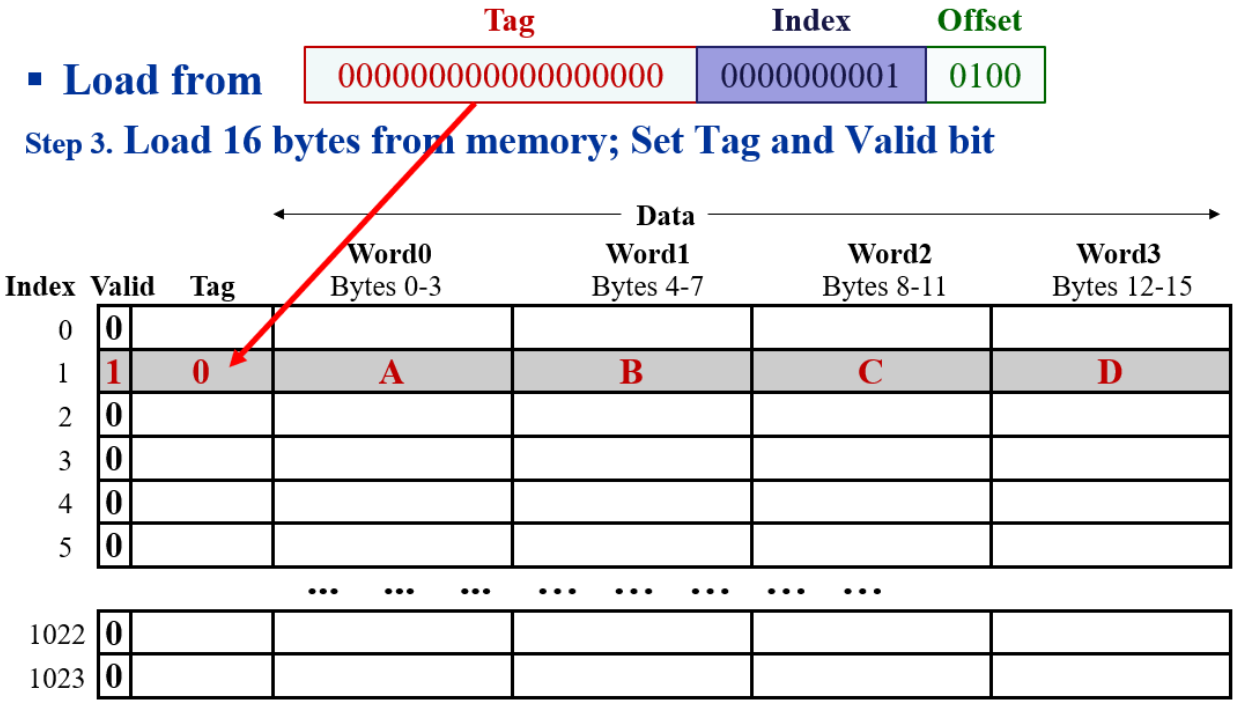
1. 所有的field都是空的
2. 所有的valid bit都为0

> 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]

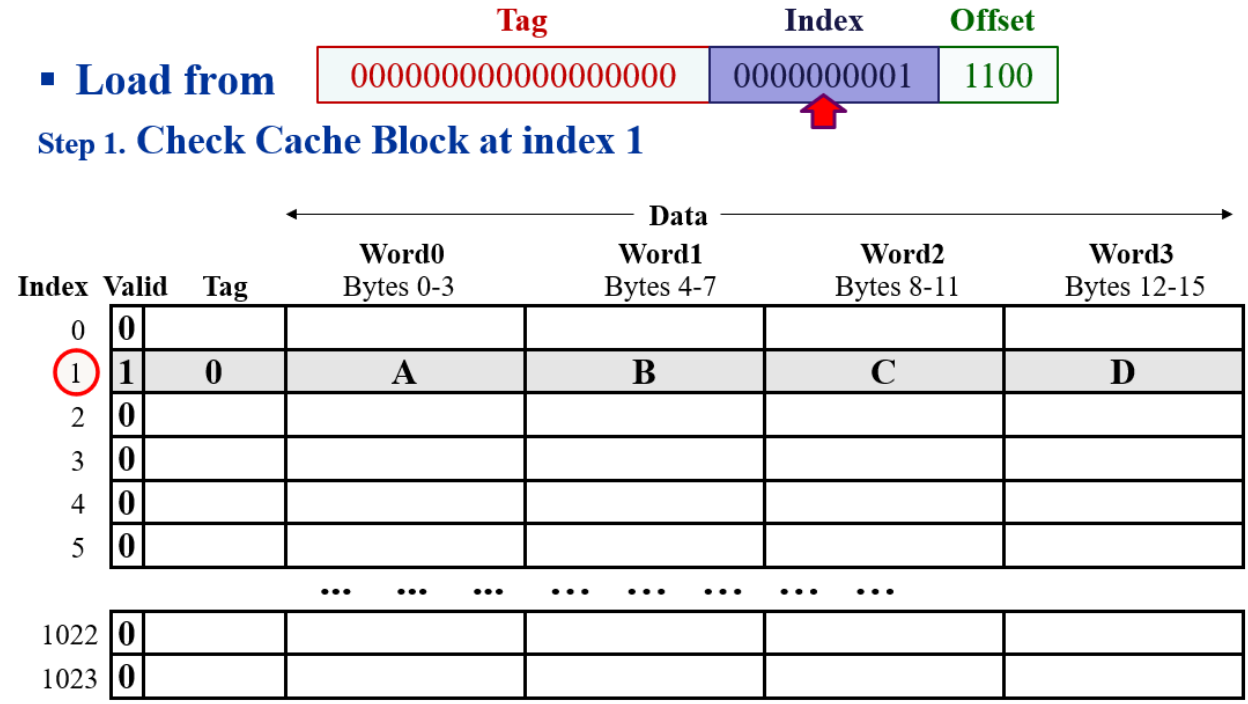




此处读取了index为1，tag为0，offset为0100的块。

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

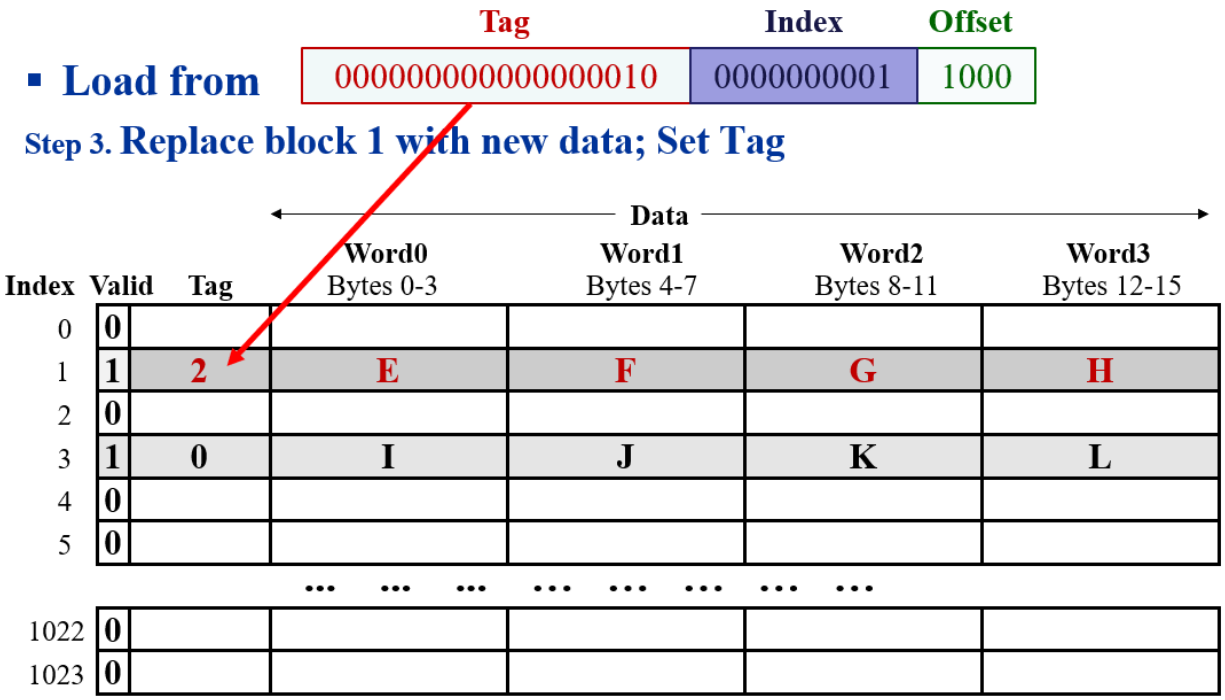
#3 – Cache hit



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

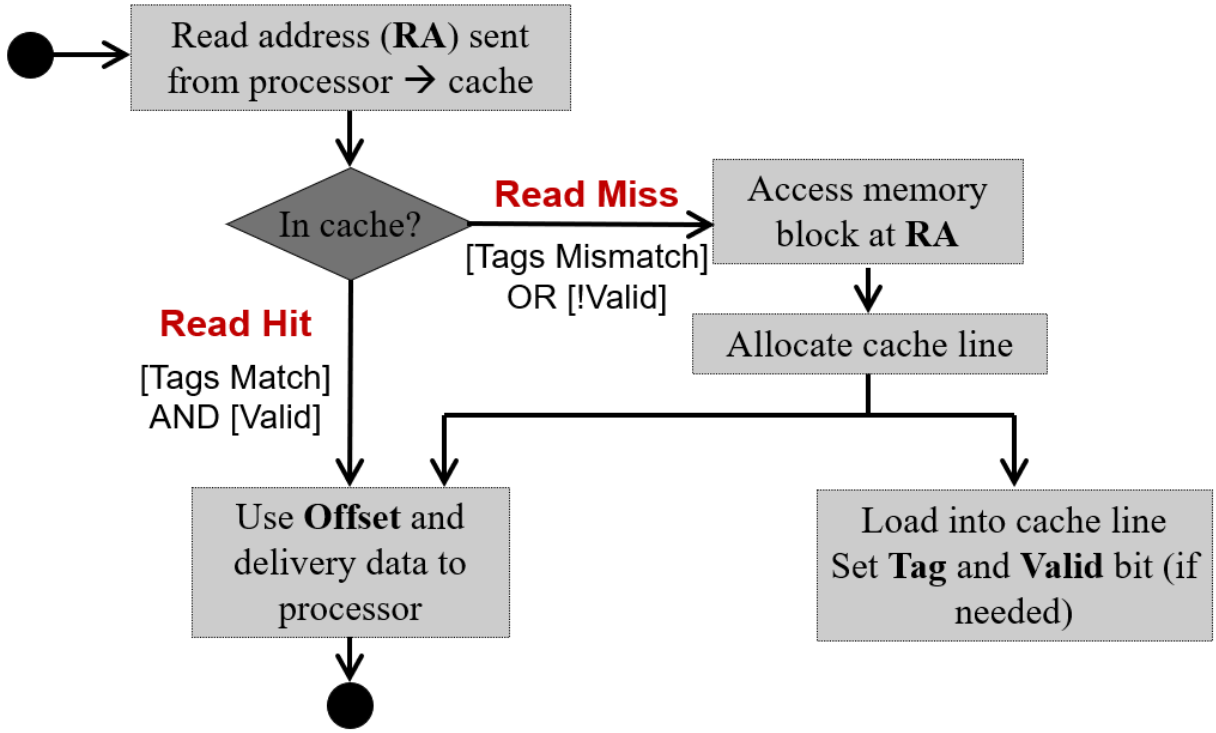
#4 - Different tag



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

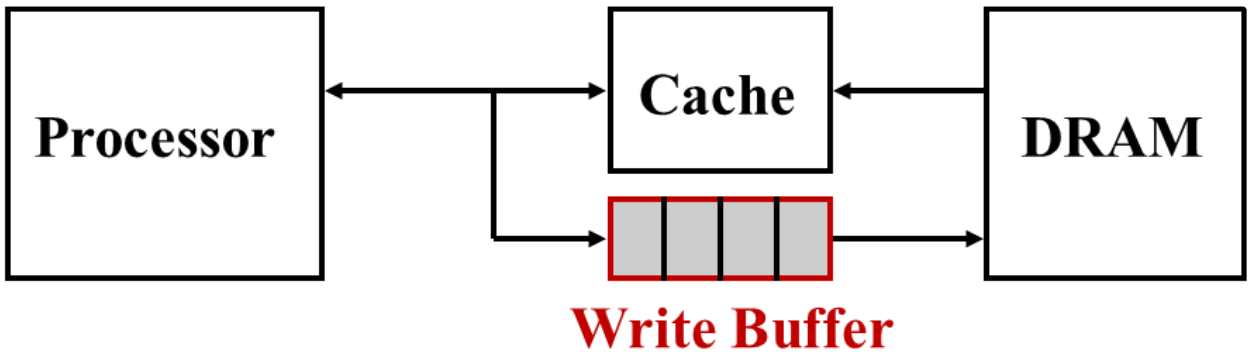
Solution 2: Write-back cache

- 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

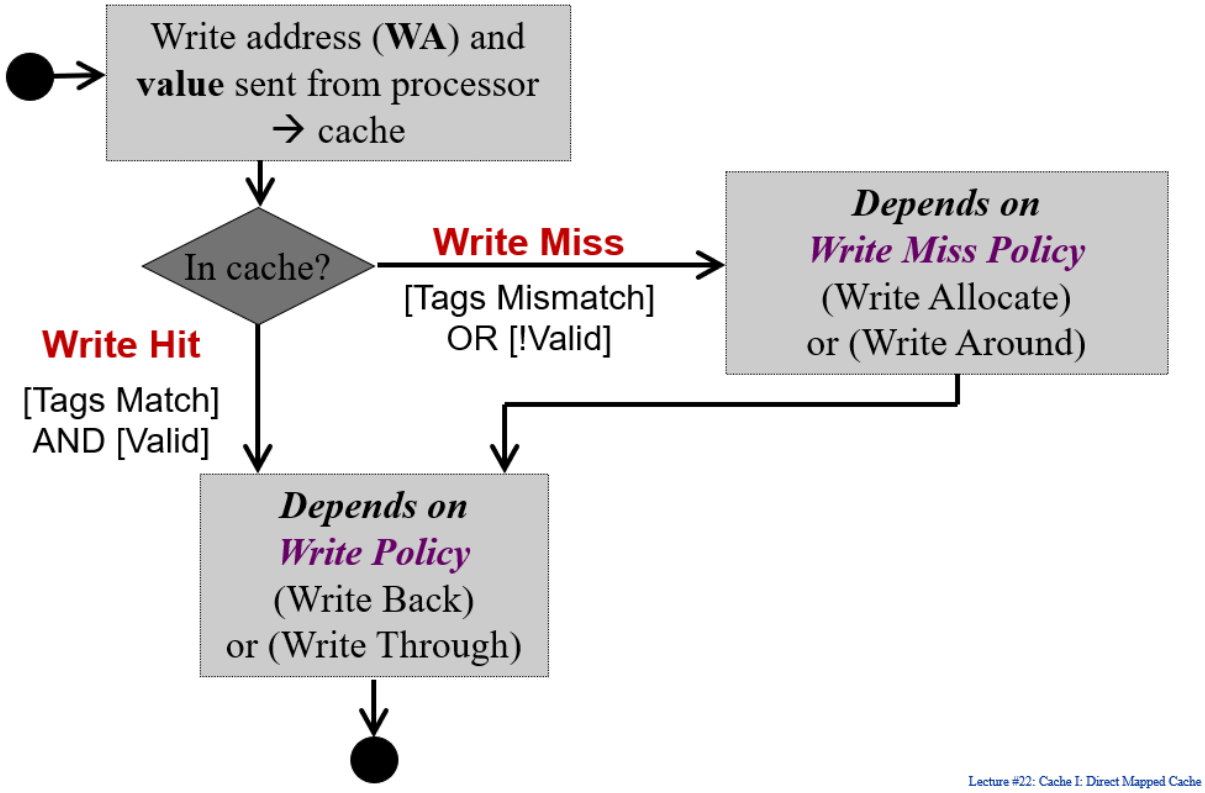
- Load the complete block into cache
- Change only the required word in cache
- Write to main memory depends on write policy (Write-through or Write-back)

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:

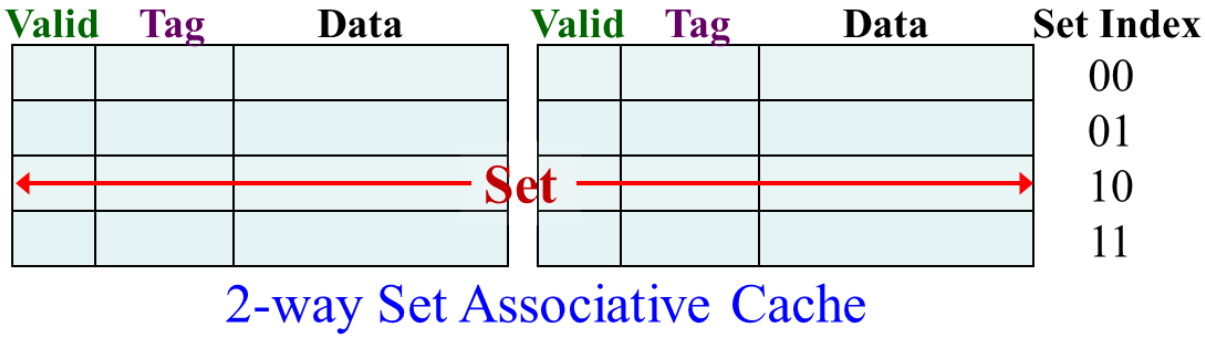
- Cache consists of a number of sets:
  - Each set contains N cache blocks
- Each memory block maps to a unique cache set
- Within the set, a memory block can be placed in any of the N cache block in the set

> 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



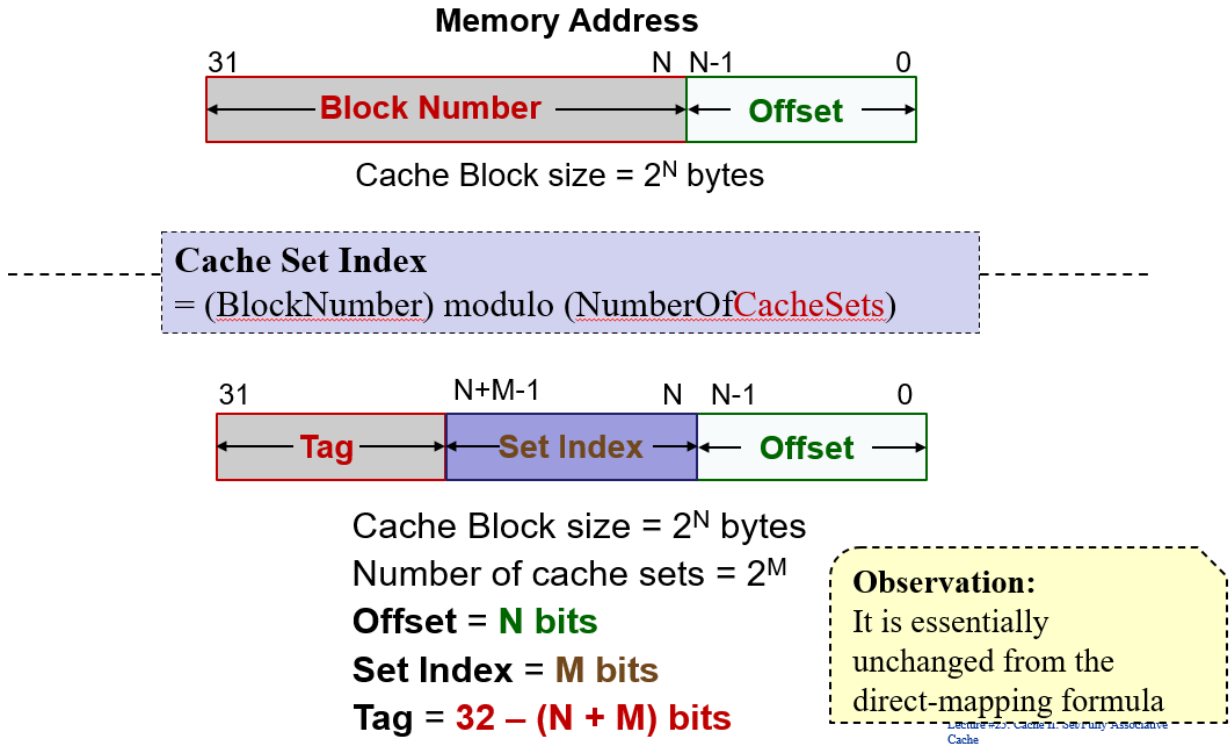
An example of 2-way set associative cache

- Each set has two cache blocks

A memory block maps to a unique set

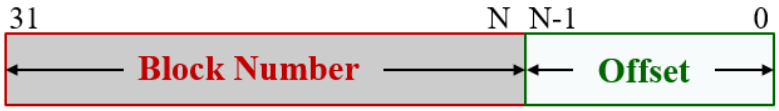
- In the set, the memory block can be placed in either of the cache blocks
- Need to search both to look for the memory block

10.8.2 Mapping



10.8.3 Example

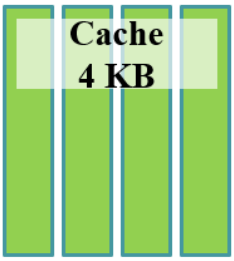
Memory Address



Offset, N = 2 bits

Block Number = 32 – 2 = **30 bits**

Check: Number of Blocks = 2^{30}



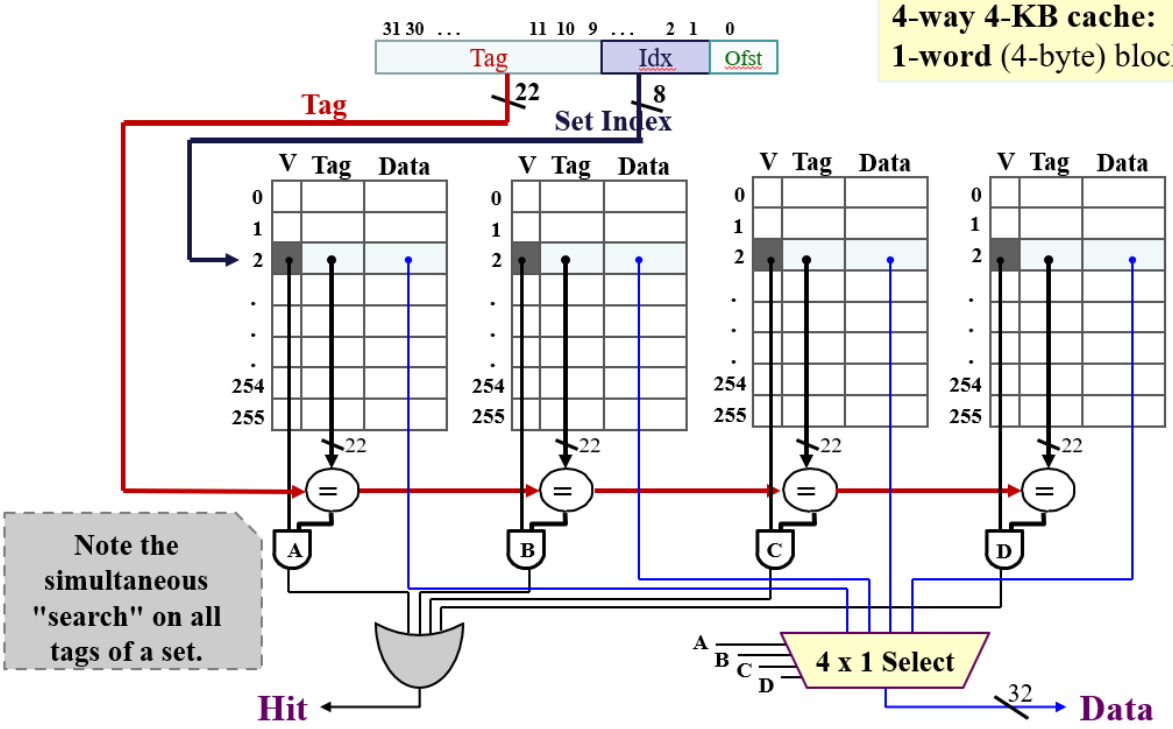
Number of Cache Blocks
= 4KB / 4bytes = 1024 = 2^{10}

4-way associative, number of sets
= 1024 / 4 = 256 = 2^{8}

Set Index, M = 8 bits

Cache Tag = 32 – 8 – 2 = **22 bits**

4-way 4-KB cache:
1-word (4-byte) blocks



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

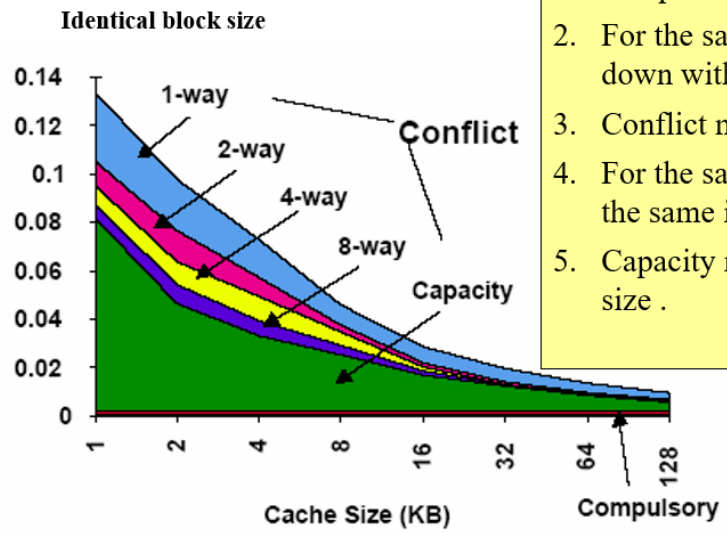
Load from `36` :

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:

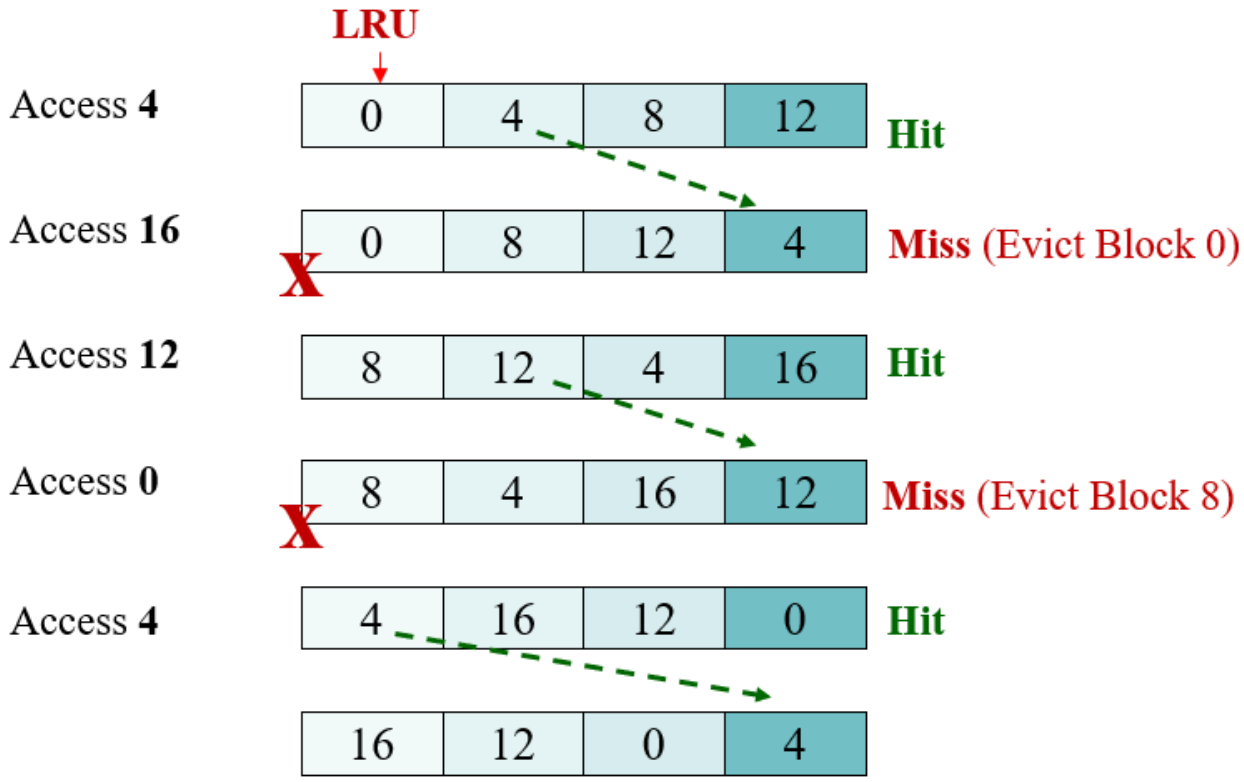
- Can choose where to place a memory block
- Potentially replacing another cache block if full
- Need block replacement policy

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

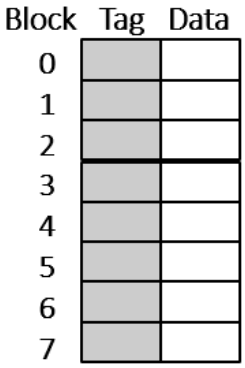
Other replacement policies:

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

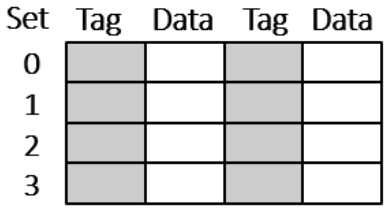
10.11 Summary

10.11.1 Cache Organizations

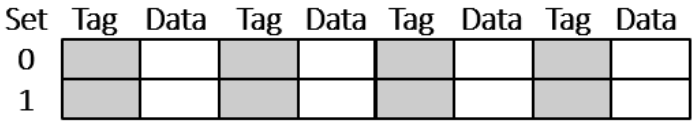
One-way set associative (direct mapped)



Two-way set associative



Four-way set associative



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