



University of
Nottingham

UK | CHINA | MALAYSIA

MIPS Programming 1

Dr. Heng Yu

AY2022-23, Spring Semester
COMP1047: Systems and Architecture
Week 3-4



- **MIPS Instruction Basics**
- **MIPS Operands**
 - Register operands and their organization
 - Memory operands and data transfer
 - Immediate operands
- **Other MIPS Operations**
 - Shift and bitwise operations
 - Floating point and multiplication/division





Learning Objectives

- Know the Von Neumann architecture used to store and execute the instructions
- Understand the syntax and know how to use the instructions taught in this lecture
- Understand registers, memory organization (addressing mode, endianness, etc.)
- Write simple MIPS programs with instruction taught in this lecture.





- **MIPS Instruction Basics**

- **MIPS Operands**

- Register operands and their organization
- Memory operands and data transfer
- Immediate operands

- **Other MIPS Operations**

- Shift and bitwise operations
- Floating point and multiplication/division

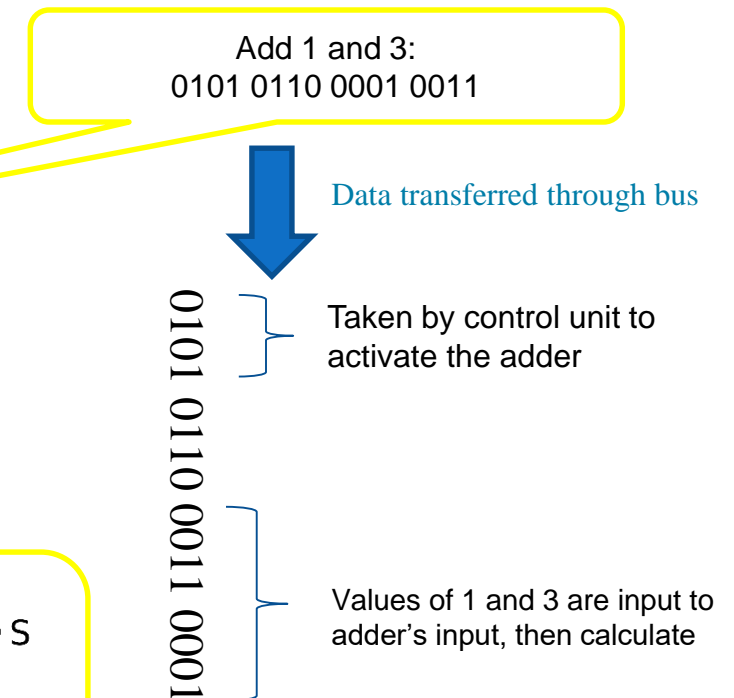
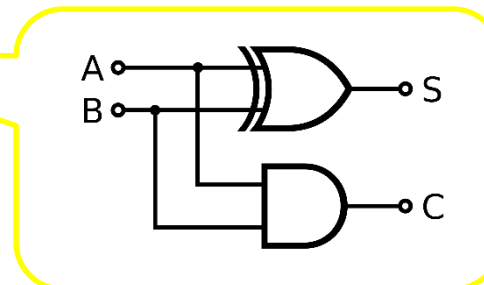
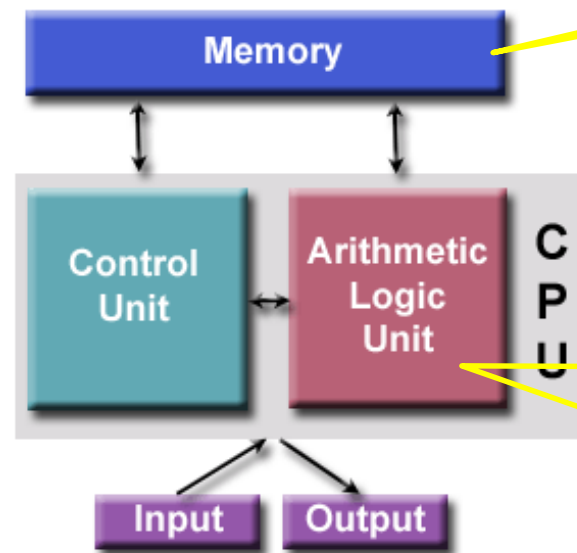




- Bit
 - Binary digit
 - Either 0 or 1
 - limited to represent two values
- Byte
 - A sequence of bits
 - Since the mid 1960's, a byte has 8 bits in length
 - 256 possible sequences
- Word
 - Amount of data computer can process in one step
 - Today most CPUs have a word size of 32 or 64 bits
 - On the 32-bit MIPS, a word is 4 bytes long

Inside the Computer Architecture

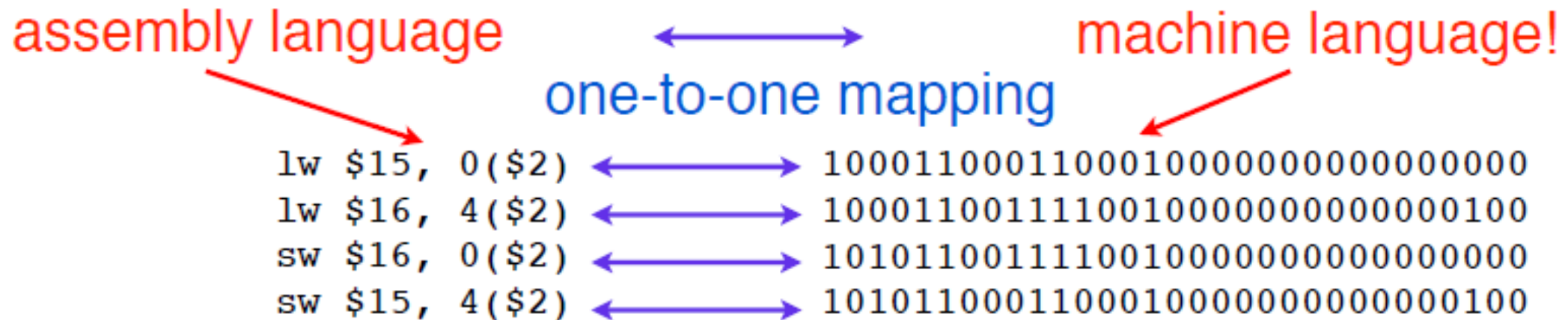
- Processor: A bunch of digital circuits that operates on 0's and 1's
- Memory: A bunch of digital circuits that stores and provides 0's and 1's for processors
- Those 0's and 1's: Instructions and Data
- Also called “stored program architecture”





Languages that communicate

- Processor (More accurately, digital circuits) understands machine languages: 100010101010100101...
- Programmer would write machine languages, but in mnemonic:
Assembly language





Assembly language

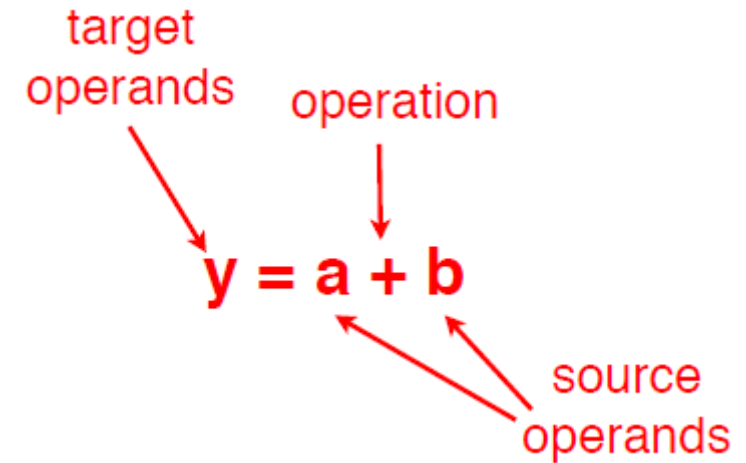
- To command a computer, you must understand its language.
 - **Instructions**: words in a computer's language
 - **Instruction set**: the vocabulary of a computer's language
- Instructions indicate the operation to perform and the operands to use.
 - **Assembly language**: human-readable format of instructions
 - **Machine language**: computer-readable format (1's and 0's)
- Recall: instructions are stored in memory

add \$r3, \$r1, \$r2



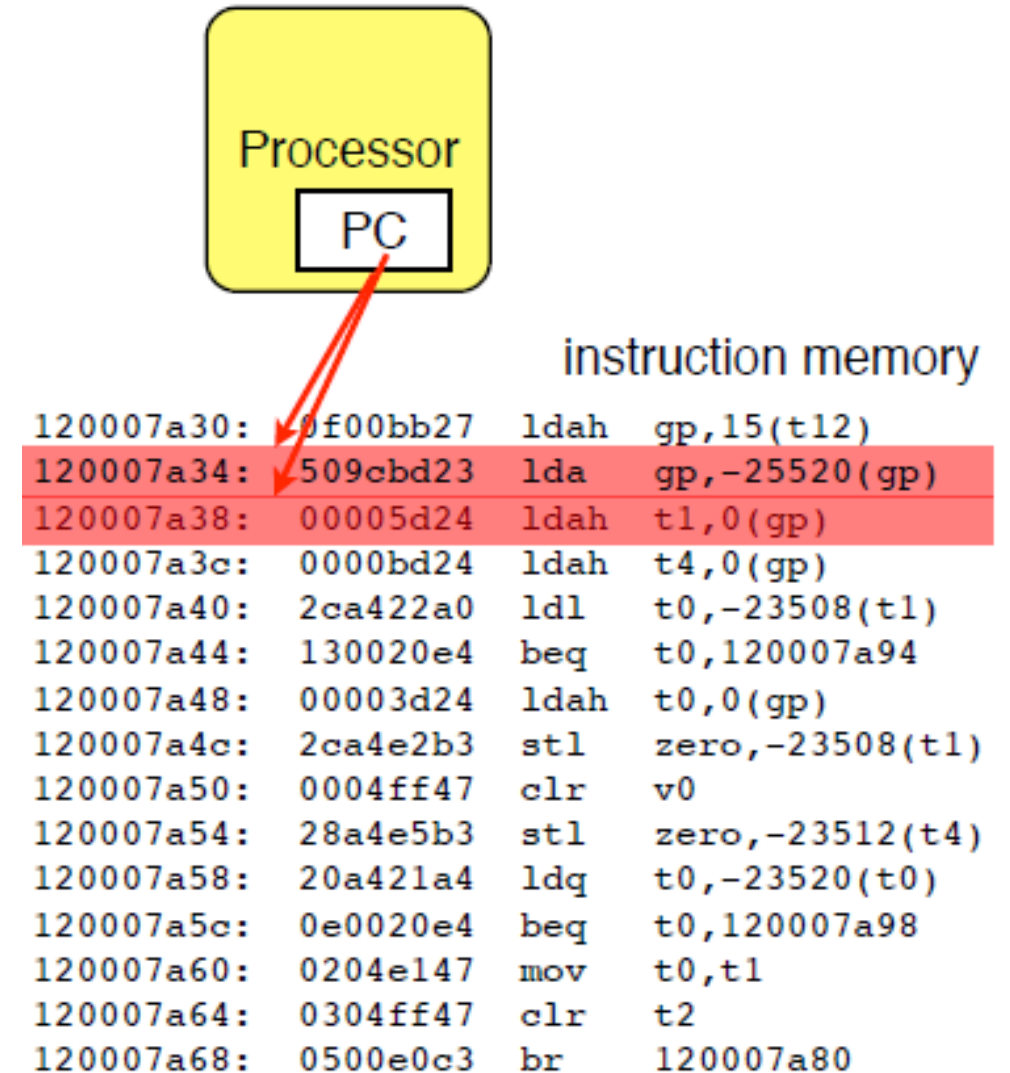
How should an instruction look like?

- Operations
 - What operations?
e.g. add, sub, mul, and etc.
 - How many operations?
- Operands
 - How many operand?
 - What type of operands?
 - Memory/register/label/number(i.e., immediate value)
- Instruction Format
 - Length? How many bits? Equal length?
 - Formats?



The stored program model

- A snapshot of instruction memory:
- Program Counter (PC)
 - Points to the memory location of the current instruction
 - Processor fetches instructions from where PC points
 - Advances/changes for the next instruction.
- PC is a piece of hardware placed in CPU.





Hexadecimal

- Base 16
 - Compact representation of bit strings
 - 4 bits per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

Example: eca8 6420

1110 1100 1010 1000 0110 0100 0010 0000

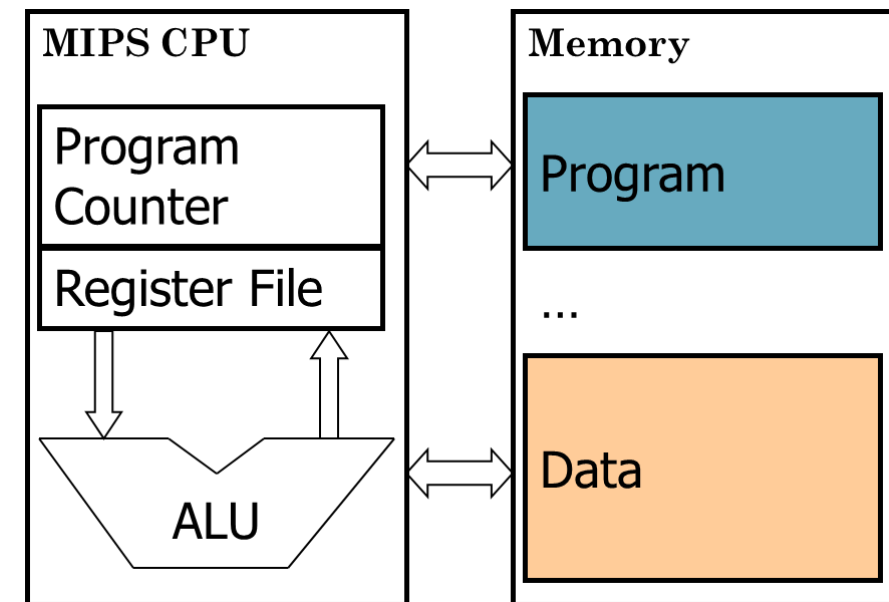


- MIPS (Microprocessor without Interlocked Processor States)
 - Instruction Set Architecture (ISA) based on Reduced Instruction Set Computing (RISC) CPU design strategy
 - Make instructions simple, but execute them fast
 - In contrast with Complex Instruction Set Computing (CISC)
 - expensive/slow memory, writing machine code difficult
 - each instruction does as much as possible
 - e.g. Intel Pentium
- 32-bit version was introduced by a team led by John L. Hennessy at Stanford University in the 1980s
 - Basic concept is to increase performance through the use of deep instruction pipelines
- 64-bit version was introduced in 1991
- Used in embedded systems like game consoles (Nintendo, Sony Playstation 1,2, PSP), graphics workstation (SGI), and Loongson

A sample MIPS program

```
msg:  .data
      .asciiz "Hello, world!\n"
      .text
      .globl main

main:  la $a0, msg      #load label msg
      li $v0, 4         #load immediate
      syscall          # print it
      li $v0, 10
      syscall          # exit
```





MIPS Basic Syntax

- **Assembler directives** begin with a dot ‘.’
 - **.data**
 - Start assembling data used by the program
 - **.text**
 - Start assembling program instructions
 - **.asciiz**
 - Place a null-terminated ASCII string in memory
- **Labels** are names followed by a colon ‘:’
 - Descriptive names chosen by programmers
 - The assembler will assign memory addresses to labels for later reference
 - The label “main” is the entry of the program
- Machine **instructions**
 - Lines after “main:” contain symbolic machine instructions
- **Comments** begin with a hash key ‘#’, until end of line

```
.data
msg:  .asciiz "Hello, world!\n"
.text
.globl main

main:
    la $a0, msg      #load label msg
    li $v0, 4         #load immediate
    syscall          # print it
    li $v0, 10
    syscall           # exit
```



- **MIPS Instruction Basics**

- **MIPS Operands**

- Register operands and their organization
- Memory operands and data transfer
- Immediate operands

- **Other MIPS Operations**

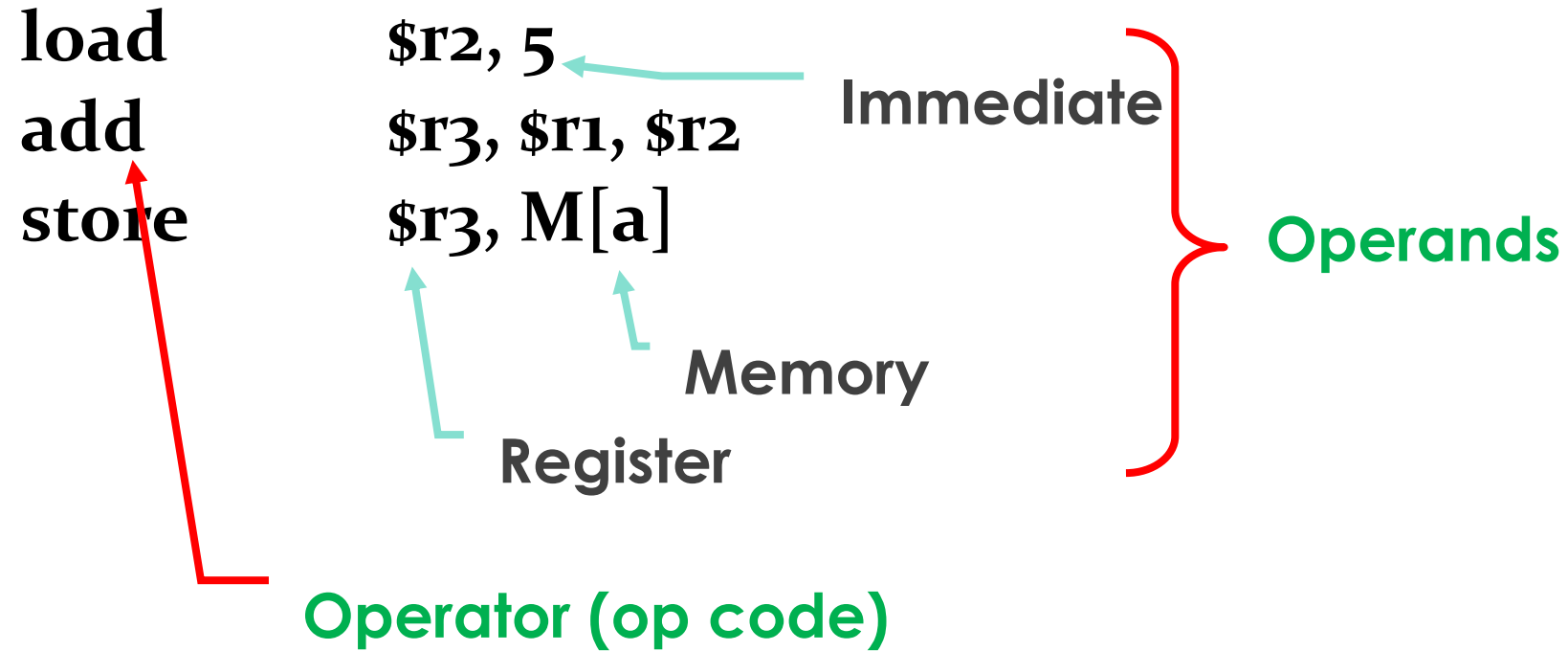
- Shift and bitwise operations
- Floating point and multiplication/division





MIPS Operands

a = b + 5;



Note: these statements are not yet MIPS instructions.

- General syntax

Three operands	Two operands	Other
op dst, src, src	op dst, src	op
op dst, src, imm	op dst, imm	op src

- dst – destination register/memory
- src – source register/memory
- imm – immediate value (16 bits)



MIPS Arithmetic: Add

Syntax of **add** instruction:

1 2 3 4
add \$s0, \$s1, \$s2

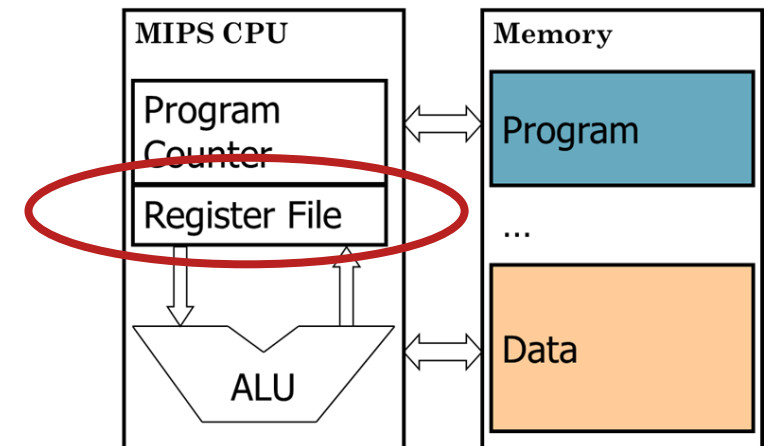
@s2
D = M
@s1
D = D + M
@s0
M = D

- 1 -> operation name
- 2 -> operand getting result (“destination”)
- 3 -> 1st operand for operation (“source1”)
- 4 -> 2nd operand for operation (“source2”)

- Add the contents of the register **rs1** and **rs2** and store the sum in the register **rd**.
- In high-level language: $c = a + b$
- Example: **add** \$s0, \$t0, \$t1

	Before	After
\$s0	4	11
\$t0	5	5
\$t1	6	6

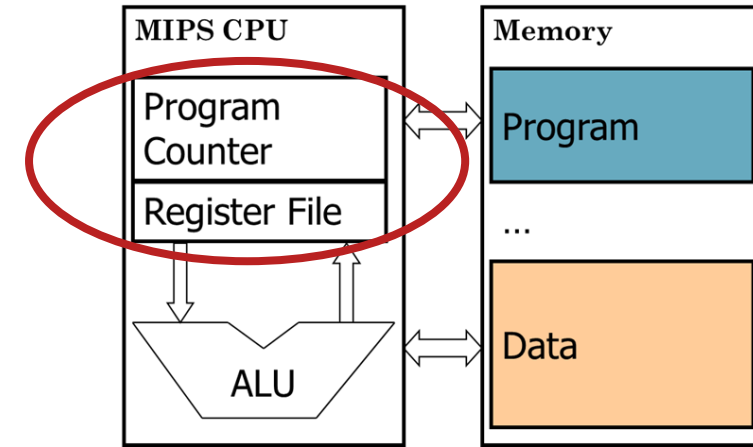
- Unlike high-level language, assembly don't use variables
=> (most) assembly operands are registers
 - ✓ Limited number (32 for MIPS) of special hardware built directly inside the CPU
 - ✓ Operations are performed on them directly
- Benefits:
 - ✓ Registers in CPU => faster than memory
 - ✓ Registers are easier for a compiler to use
 - ✓ e.g., as a place for temporary storage
 - ✓ Registers can hold variables to reduce memory traffic



MIPS Registers

- 32 registers, each is 32 bits wide
 - Groups of 32 bits called a *word* in MIPS
 - Registers are numbered from 0 to 31
 - Each can be referred to by number or name
 - Number references:
\$0, \$1, \$2, ... \$30, \$31
 - By convention, each register also has a name to make it easier to code, e.g.,

\$16 – \$23	➔	\$s0 – \$s7	(for variables)
\$8 – \$15	➔	\$t0 – \$t7	(for temporary)
- 32 x 32-bit FP registers
- Others: PC, etc.





MIPS Register Conventions

0	zero	constant 0
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments
5	a1	
6	a2	
7	a3	
8	t0	temporary: caller saves
...		(callee can clobber)
15	t7	
16	s0	callee saves
...		(caller can clobber)
23	s7	
24	t8	temporary (cont'd)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	pointer to global area
29	sp	stack pointer
30	fp	frame pointer
31	ra	return address (HW)



MIPS Arithmetic: Subtract

`sub rd, rs1, rs2`

- Subtract the contents of the register `rs1` and `rs2` and store the sum in the register `rd`.
- In high-level language: $c = a - b$
- Example: `sub $s0, $t0, $t1`

	Before	After
<code>\$s0</code>	4	-1
<code>\$t0</code>	5	5
<code>\$t1</code>	6	6



- Given the C statement:

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

- **Question: How do we translate it into assembly language?**

$g: t0, h: t1, i: t2, j: t3$

Add s0, t0, t1

Add s1, t2, t3

Sub s2, s0, s1

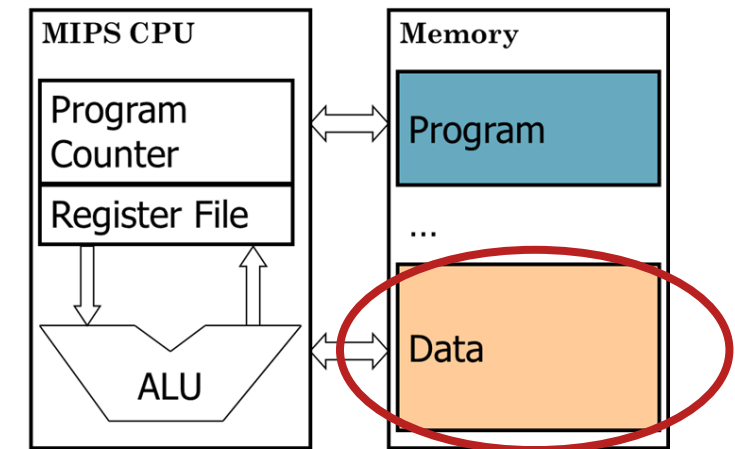


- **MIPS Instruction Basics**
- **MIPS Operands**
 - Register operands and their organization
 - **Memory operands and data transfer**
 - Immediate operands
- **Other MIPS Operations**
 - Shift and bitwise operations
 - Floating point and multiplication/division



Memory Operands

- C/Java variables map onto registers; what about large data structures like arrays?
 - Memory contains such data structures
- But MIPS arithmetic instructions operate on registers, not directly on memory
 - Data transfer instructions (`lw`, `sw`, `lb`, `sb`, etc.) to transfer between memory and register.



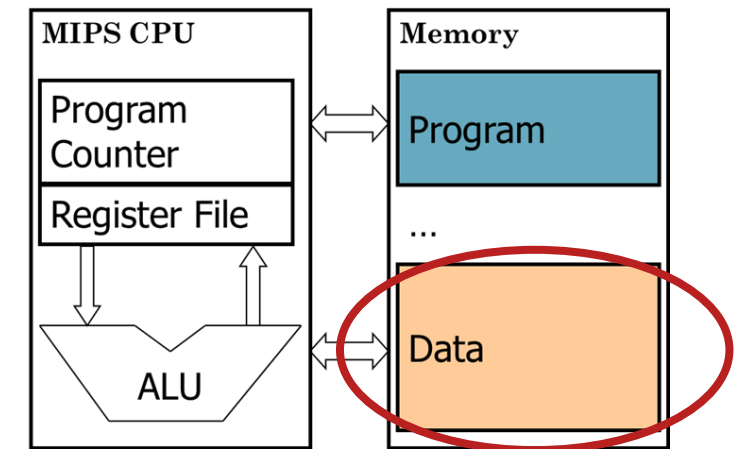
`lw`: load a word

`sw`: store a word, reverse operation of `lw`

`lb`: load byte, `sb`: save byte

Data Transfer: Memory to Register

- To transfer a word of data, need to specify two things:
 - Register: specify which register to send the data
 - By number (0 - 31)
 - Memory address:
 - Think of memory as a 1D array
 - Address it by supplying a pointer to a memory address
 - Offset (in bytes) from this pointer
 - The desired memory address is the sum of these two values, e.g., 8(\$t0)
 - Specifies the memory address pointed to by the value in \$t0, plus 8 bytes.
 - Each address is 32 bits

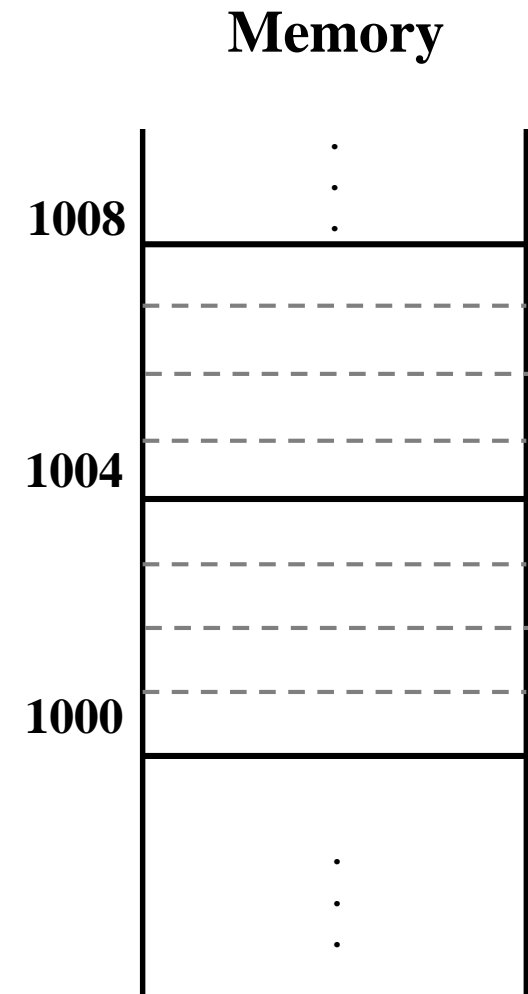




Data Transfer: Memory to Register

1 word = 4 bytes

1111 0000 1111 0001 1111 0010 1111 0011





Data Transfer: Memory to Register

- Load Instruction Syntax:

1 **2** **3** **4**
`lw $t0, 12 ($s0)`

1 operation name (op code)

2 register that will receive value

3 numerical offset in bytes

4 register containing pointer to memory

- Example: `lw $t0, 12 ($s0)`
 - lw (Load Word, so a word (32 bits) is loaded at a time)
 - Take the pointer in \$s0, add 12 bytes to it, and then load the value from the memory pointed by this calculated sum into register \$t0
- Notes:
 - \$s0 is called the *base register*, 12 is called the *offset*
 - Offset is generally used in accessing elements of array
 - Base register points to the beginning of the array



Data Transfer: Memory to Register

- $\$s0 = 1000$ = word 0

$$12 = 4 \text{ byte} * 3$$

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

- $\$t0 = ?$ 999

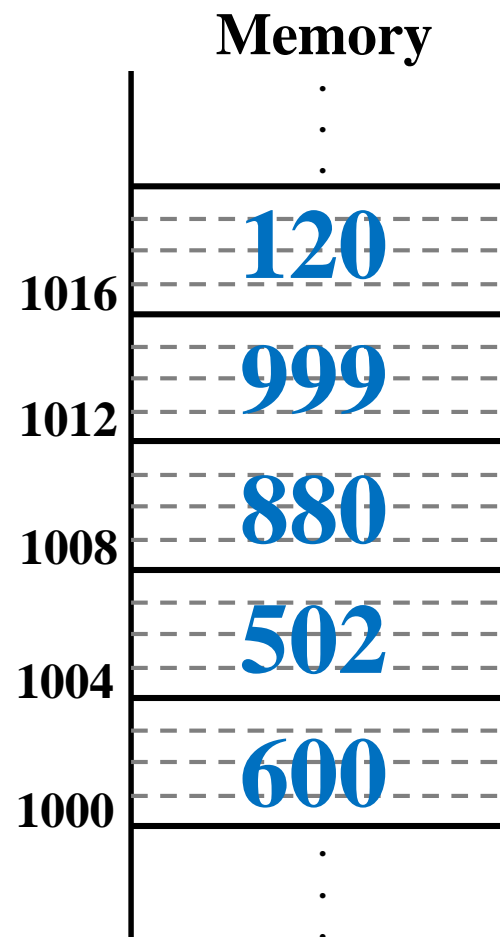
12(\$s0) 相当于对s0的地址进行加法

word 3

word 2

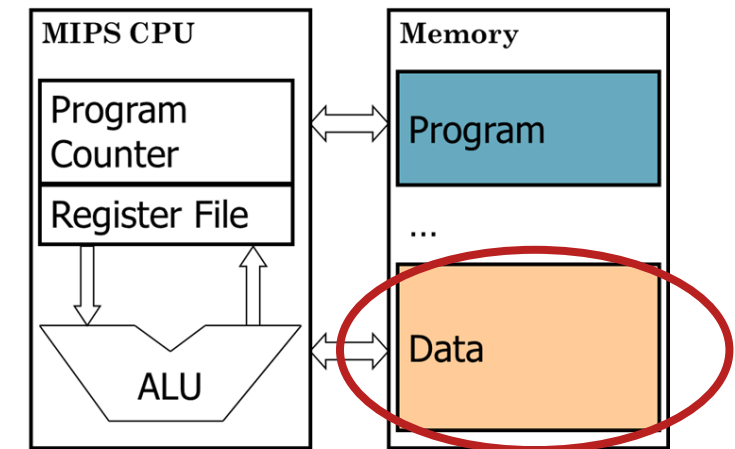
word 1

word 0



Data Transfer: Register to Memory

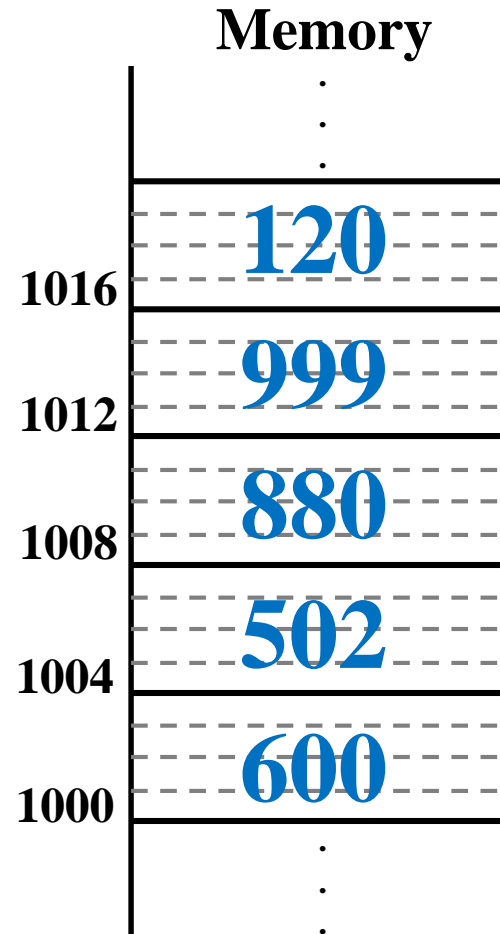
- Also want to store value from a register into memory
- Store instruction syntax is identical to Load instruction syntax
- Example: `sw $t0, 12($s0)`
 - sw (Store Word, a word (32 bits) is stored at a time)
 - This instruction will take the pointer in \$s0, add 12 bytes to it, and then store the value from register \$t0 into the memory address pointed to by the calculated sum





Data Transfer: Memory to Register

- $\$s0 = 1000$
- $\$t0 = 25$
- $\text{sw } \$t0, 12(\$s0)$
- $M[1012] = 25$





Memory Addressing

- Only load and store instructions can access memory
- byte, half words, words are aligned

Byte addresses

Address	Data
0x0000	0xAA
0x0001	0x15
0x0002	0x13
0x0003	0xFF
0x0004	0x76
...	.
0xFFFFE	.
0xFFFFF	.

Half Word Addresses

Address	Data
0x0000	0xAA15
0x0002	0x13FF
0x0004	.
0x0006	.
...	.
...	.
...	.
0xFFFC	.

Word Addresses

Address	Data
0x0000	0xAA1513FF
0x0004	.
0x0008	.
0x000C	.
...	.
...	.
...	.
0xFFFC	.



Byte-Addressable Memory

- MIPS memory is byte-addressable (not word addressable)
- Each byte has a unique address
- Load and store single bytes: **load byte (lb)** and **store byte (sb)**
- Each 32-bit words has 4 bytes, so the word address increments by 4

Word Address	Data								
⋮	⋮								⋮
0000000C	4	0	F	3	0	7	8	8	Word 3
00000008	0	1	E	E	2	8	4	2	Word 2
00000004	F	2	F	1	A	C	0	7	Word 1
00000000	A	B	C	D	E	F	7	8	Word 0

width = 4 bytes



Reading Byte-Addressable Memory

- The address of a memory word must now be multiplied by 4. For example,
 - the **address** of memory **word 2** is $2 \times 4 = 8$
 - the address of memory **word 10** is $10 \times 4 = 40$ (**0x28**)
- Load a word of data (word 1) at memory address 4 into \$s3.
- \$s3 holds the value 0xF2F1AC07 after the instruction completes.

MIPS assembly code

```
lw $s3, 4($0)    # read memory word 1 into $s3
```

Word Address	Data								
⋮	⋮								⋮
0000000C	4	0	F	3	0	7	8	8	Word 3
00000008	0	1	E	E	2	8	4	2	Word 2
00000004	F	2	F	1	A	C	0	7	Word 1
00000000	A	B	C	D	E	F	7	8	Word 0

← width = 4 bytes →



Writing Byte-Addressable Memory

The assembly code below stores the value held in \$t7 into memory address 0x2C (44).

- \$t7 = 0xF2F1AC07

MIPS assembly code

```
sw $t7, 44($0)  # write $t7 into memory word 11
```

Word Address	Data	
⋮	⋮	⋮
00000034	4 0 F 3 0 7 8 8	Wd 13
00000030	0 1 E E 2 8 4 2	Wd 12
0000002C	F 2 F 1 A C 0 7	Wd 11
00000028	A B C D E F 7 8	Wd 10

← width = 4 bytes →

lw and sw use bytes in offset!!!



Big-Endian and Little-Endian Memory

- How to order the bytes within a word?
- Word address is the same for big- or little-endian
- Little-endian: orders bytes **starting at the little (least significant) end** (e.g., Intel IA-32, some MIPS CPUs)
- Big-endian: order bytes **starting at the big (most significant) end** (e.g., IBM PowerPC, some MIPS CPUs)

Big-Endian

Byte Address			
⋮			
C	D	E	F
8	9	A	B
4	5	6	7
0	1	2	3
MSB		LSB	

Little-Endian

Byte Address			
⋮			
F	E	D	C
B	A	9	8
7	6	5	4
3	2	1	0
MSB		LSB	



Writing Byte-Addressable Memory

The assembly code below stores the value held in \$t7 into memory address 0x2C (44).

- \$t7 = 0xF2F1AC07

MIPS assembly code

```
sw $t7, 44($0)  # write $t7 into memory word 11
```

Word Address	Data	
⋮	⋮	⋮
00000034	4 0 F 3 0 7 8 8	Wd 13
00000030	0 1 E E 2 8 4 2	Wd 12
0000002C	0 7 A C F 1 F 2	Wd 11
00000028	A B C D E F 7 8	Wd 10

width = 4 bytes

This is a big-endian memory.
How about little-endian?



Role of Registers vs. Memory

- What if there are more variables than registers in your code?
 - Compiler tries to keep most frequently used variables in registers
 - Puts less common variables into memory: [spilling](#)
- Why not keep all variables in memory?
 - **Design Principle: Smaller is faster**
 - Registers are faster than memory
 - Registers are more versatile:
 - MIPS arithmetic instructions can read 2 registers, operate on them, and write to 1 register per instruction
 - MIPS data transfers only read or write 1 register per instruction



- **MIPS Instruction Basics**

- **MIPS Operands**

- Register operands and their organization
- Memory operands and data transfer
- **Immediate operands**

- **Other MIPS Operations**

- Shift and bitwise operations
- Floating point and multiplication/division





Constants

- Small constants are used frequently (50% of operands)

e.g., $A = A + 5;$
 $B = B + 1;$
 $C = C - 18;$

- Constant data specified/hardwired in an instruction:

```
addi $29, $29, 4  
slti $8, $18, 10  
andi $29, $29, 6  
ori $29, $29, 4
```



Immediate Operands

- Immediate: *numerical constants*
 - Often appear in code, so there are special instructions for them
 - Add Immediate:

f = g + 10 (in Java)

addi \$s0,\$s1,10 (in MIPS)

where \$s0,\$s1 are associated with f,g

- Syntax similar to add instruction, except that **the last argument is a number** instead of a register
- **No subtract immediate instruction**
 - Just use a negative constant **use addition to perform subtraction**

addi \$s2, \$s1, -1



The Constant Zero

- The number zero (0), appears very often in code; so we define **register zero to be constant 0**
- MIPS register 0 (\$zero) is the constant 0
 - Cannot be overwritten
 - This is defined in hardware, so an instruction like

addi \$0, \$0, 5 will not do anything

- Useful for common operations
 - E.g., move between registers

add \$t2, \$s1, \$zero



- **MIPS Instruction Basics**
- **MIPS Operands**
 - Register operands and their organization
 - Memory operands and data transfer
 - Immediate operands
- **Other MIPS Operations**
 - Shift and bitwise operations
 - Floating point and multiplication/division





- Up until now, we've done arithmetic (`add`, `sub`, `addi`) and memory access (`lw` and `sw`)
- All of these instructions view contents of register as a single quantity (such as a signed or unsigned integer)
- New perspective: View contents of register **as 32 bits** rather than as a single 32-bit number
- Operate **bit-wise**.
- Introduce two new classes of instructions:
 - Shift instructions
 - Logical operators



Logical Operations

- Instructions for bitwise manipulation
- Useful for extracting and inserting groups of bits in a word

Operation	C	Java	MIPS
Shift left	<<	<<	sll
Shift right	>>	>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR			or, ori
Bitwise NOT	~	~	Nor \$0



Shift Instructions

- Shift Instruction Syntax:

1 **2** **3** **4**
`sll` `$t2,` `$s0,` `4`

1 operation name (op code)

2 register that will receive value

3 first operand (register)

4 shift amount (constant)

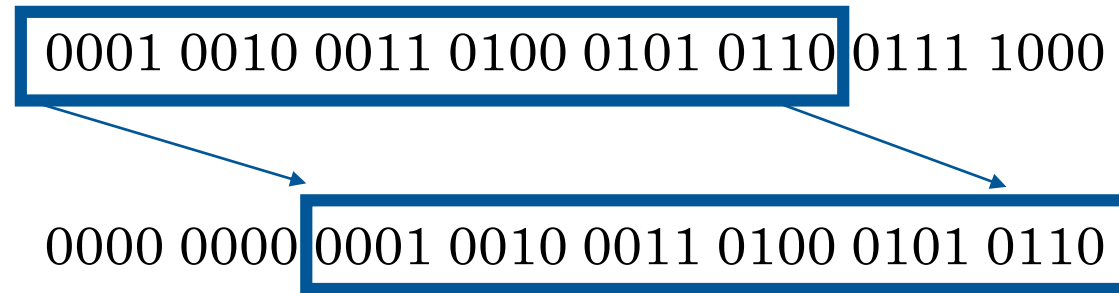
- MIPS has three shift instructions:

- `sll` (shift left logical): shifts left, fills empties with 0s
- `srl` (shift right logical): shifts right, fills empties with 0s
- `sra` (shift right arithmetic): shifts right, fills empties by sign extending

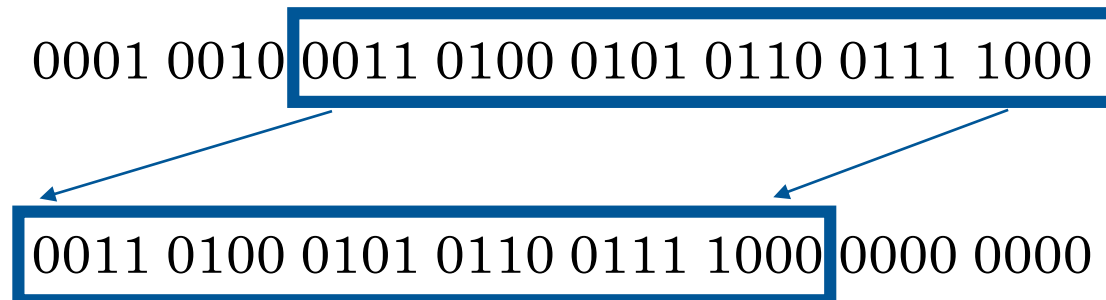


Shift Instructions

- **sll, srl**: Move (shift) all the bits in a word to the left or right by a number of bits, filling the emptied bits with 0s.
- Example: shift right logic by 8 bits



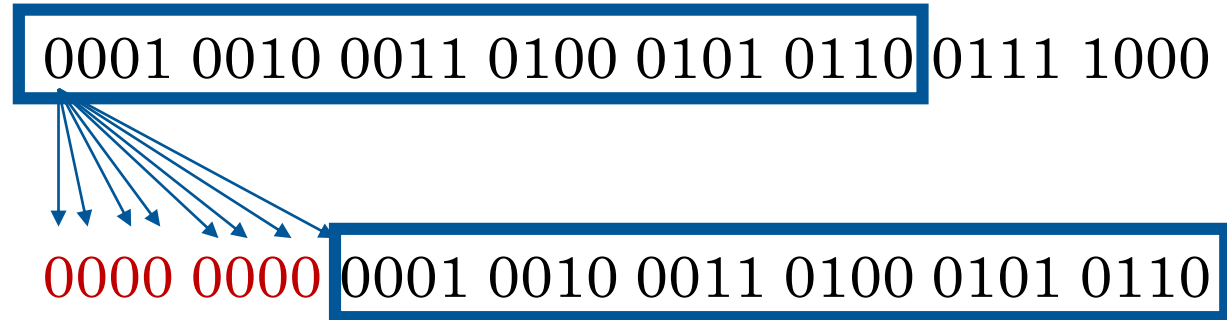
- Example: shift left by 8 bits



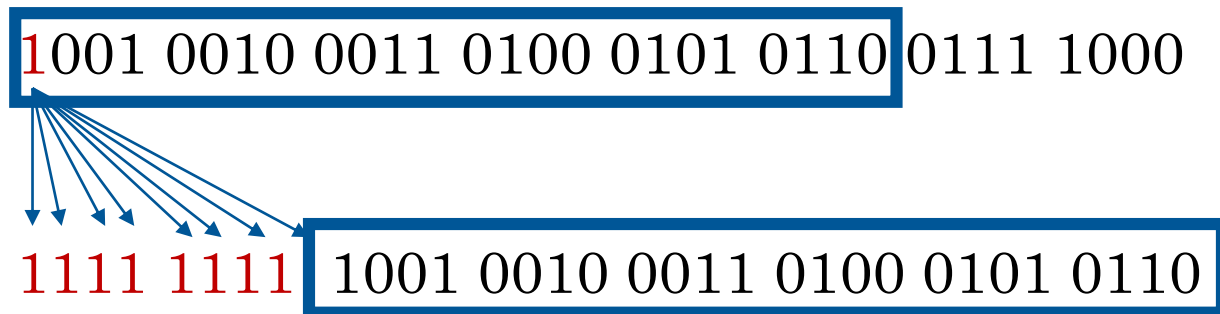


Shift Instructions

- **sra** example: shift right arithmetic by 8 bits



- Example: shift right arithmetic by 8 bits





Shift Instructions

- Shift for multiplication: in binary
 - Multiplying by 4 is same as shifting left by 2:
 - $11_2 \times 100_2 = 1100_2$
 - $1010_2 \times 100_2 = 101000_2$
 - Multiplying by 2^n is same as shifting left by n
- Since shifting is so much faster than multiplication (you can imagine how complicated multiplication is), a good compiler usually notices when C/Java code multiplies by a power of 2 and compiles it to a shift instruction:

`a *= 8;` (in C)

would compile to:

`sll $s0, $s0, 3` (in MIPS)

3 means 2^3

$$\begin{aligned} a*6 &= a*4 + a*2 \\ &= a(\text{left shift } 2) + a(\text{left shift } 1) \end{aligned}$$



AND operations

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

and \$t0, \$t1, \$t2

\$t2	0000 0000 0000 0000 0000 1101 1100 0000
\$t1	0000 0000 0000 0000 0011 1100 0000 0000
\$t0	0000 0000 0000 0000 0000 1100 0000 0000



OR operations

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

or \$t0, \$t1, \$t2

\$t2	0000	0000	0000	0000	0000	11	01	1100	0000
\$t1	0000	0000	0000	0000	00	11	1100	0000	0000
\$t0	0000	0000	0000	0000	00	11	1101	1100	0000



NOT operations

- Useful to invert bits in a word
 - Change 0 to 1, and 1 to 0
- MIPS uses NOR, a 3-operand instruction
 - $a \text{ NOR } b == \text{NOT} (a \text{ OR } b)$

nor \$t0, \$t1, \$zero

\$t1 0000 0000 0000 0000 0011 1100 0000 0000

\$t0 1111 1111 1111 1111 1100 0011 1111 1111



Bitwise operations

- Question: `and $s0, $s1, $s2` (assuming 8-bit reg.)

Registers	Before	After
\$s0	1010 1110	?
\$s1	1011 1010	?
\$s2	0011 1001	?



Bitwise operations

- E.g. `nor $s0, $s1, $s2` (assuming 8-bit reg.)

Registers	Before	After
\$s0	1010 1110	?
\$t0	1011 1010	1011 1010
\$t1	0011 1001	0011 1001

- Note there is no `norl` operation



- **MIPS Instruction Basics**

- **MIPS Operands**

- Register operands and their organization
- Memory operands and data transfer
- Immediate operands

- **Other MIPS Operations**

- Shift and bitwise operations
- Floating point and multiplication/division





MIPS Floating-Point Architecture

- IEEE operations performed by **Floating Point Unit (FPU)**
 - **MIPS core refers to FPU as coprocessor 1**
- FPU features **32 single precision (32-bit) registers**
 - \$f0, \$f1, \$f2, . . . , \$f31
- Or as **16 pairs of double precision (64-bit) registers**
 - \$f0, \$f2, \$f4, . . . , \$f30 (even registers only!)
 - Here \$fi actually stands for the pair \$fi and \$f(i + 1)
- \$f0 is **NOT** hardwired to the value 0.0!



MIPS Floating-Point Instructions

- MIPS supports IEEE 754 both single precision and double precision
- General format
 - `op.s` : single precision
 - `op.d`: double precision
- Addition
 - `add.s`: addition single
 - `add.d`: addition double
 - Format: `add.s(d) fd, fs, ft`
 - E.g. `add.s $f0, $f1, $f2`
`add.d $f0, $f2, $f4`



Load/Store for Floating Point

- Similar to `lw` and `sw` instructions, but for floating point numbers.
- Can use assembler directives: `.float num` or `.double num` to initialize the numbers to be loaded or stored.
- `lwc1 fd, n(rs)` – load word coprocessor 1
 - Load a 32-bit word at address `rs+n` into register `fd`
- `swc1 fd, n(rs)` – store word coprocessor 1
 - Store the content of register `fd` at the address given by register `rs+n`.
 - Address `rs+n` must be word aligned!
- To store/load doubles we have to execute two instructions, e.g.
 - `lwc1 $f0, 0($s0)`
 - `lwc1 $f1, 4($s0)`
 - Or with pseudo-instruction: `l.d $f0, 0($s0)`



Copying Data Between FPU and CPU

- Only **the bit pattern** is copied, not the actual value that it represents
- **mfc1 rd, fs** – move from coprocessor 1 fs to register rd
 - `mfc1 $t0, $f7` --- copy content of \$f7 to \$t0 在\$t0中储存的是IEEE754形式的1并不是数值1
- **mtc1 rs, fd** – move to coprocessor 1
 - `mtc1 $zero, $f12` --- set \$f12 := 0

Other floating point instructions, such as subtraction, multiply, comparison, etc., please refer to the textbook if needed to use.



Multiplication Instructions

- MIPS stores the 64 bit result of the multiplication of two 32 bit registers in two special 32-bit registers **Hi** and **Lo**
 - Bits 32 to 63 are stored in **Hi**
 - Bits 0 to 31 are stored in **Lo**
- There are two instructions: multiply (`mult`) and unsigned multiply (`multu`)
- e.g. `mult $s1, $s2`
 - Performs a signed multiplication of the registers `$s1` and `$s2`, storing the result in **Hi** and **Lo**
- We can use the instructions: move from low (`mflw`) and move from high (`mflh`), e.g. `mflw $s0`
 - Store the lower 32-bit of the result of the previous multiplication operation in register `$s0`



Multiplication Instructions

- Two small integers are multiplied. Where is the result?
- If the result is small enough all the significant bits will be contained in **Lo** and **Hi** will contain all zeros
- Since it is common that we are only interested in the 32 bit result of a multiplication, the sequence

```
mult $s1, $s2
mflo $s0
```

can be encoded as one operation:

```
mul $s0, $s1, $s2
```
- Note `mul` does not check for overflow, but a pseudo-instruction `multiply with overflow` `mulo` does



Division Instructions

- Recall some terminology: if we divide one positive integer by another, say **78/21**, or more generally “**dividend/divisor**” then we get a quotient and a remainder

$$\textit{dividend} = \textit{quotient} \times \textit{divisor} + \textit{remainder}$$
$$78 = 3 \times 21 + 15$$

- MIPS stores the result of the division of two 32 bit registers in two special 32-bit registers **Hi** and **Lo**
 - remainder is stored in **Hi**
 - quotient is stored in **Lo**
- There are two instructions: divide (**div**) and unsigned divide (**divu**)



Division Instructions

- There are two instructions: divide (`div`) and unsigned divide (`divu`)
- e.g. `div $s1, $s2`
 - Performs a signed division of the registers `$s1` and `$s2`, storing the remainder in **Hi** and quotient in **Lo**
- We can use the instructions: move from low (`mflo`) and move from high (`mfhi`), e.g. `mflo $s0`
 - Store the quotient of the previous operation in register `$s0`
- If we use `div`, `divu` with 3 arguments the assembler interprets this as a pseudo instruction and generates `mflo` automatically
- i.e. The instruction: `div $s0, $s1, $s2` corresponds to
`div $s1, $s2`
`mflo $s0`



- **Stored Program Model**
- **Three types of MIPS operands**
 - Register, Memory, Immediate
 - Memory organizations
- **MIPS Instructions**
 - Addition, subtraction, immediate, load/store, bitwise, shift, multiplication, division, floating point, etc.
 - Hex representation of binary numbers.





University of
Nottingham

UK | CHINA | MALAYSIA

Stay Tuned.