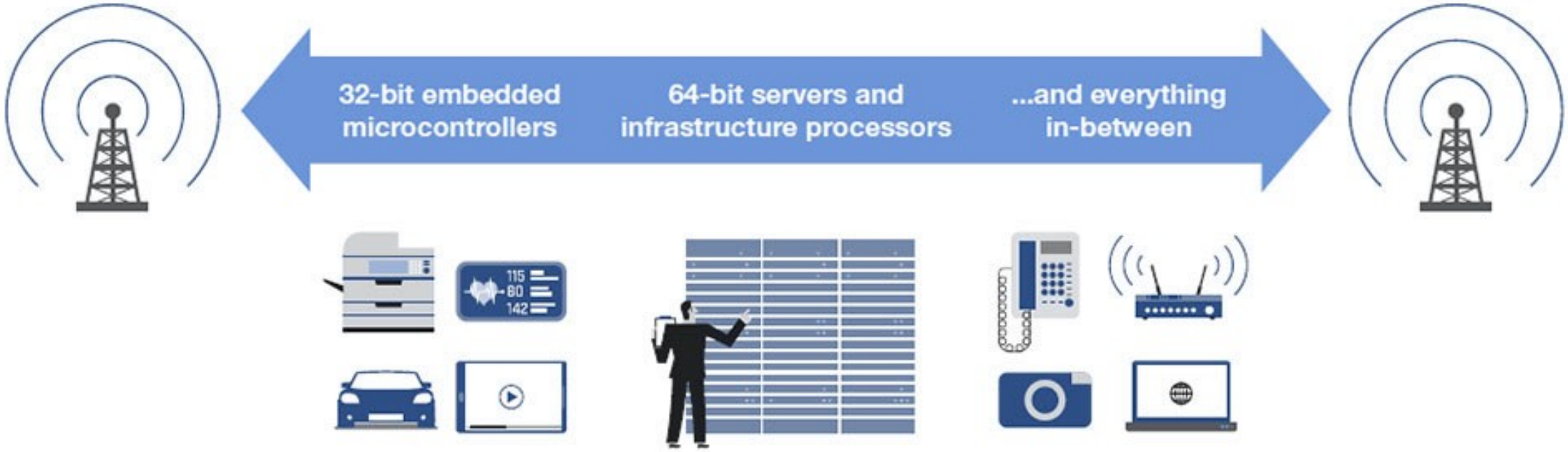


MIPS Instruction Set Architecture

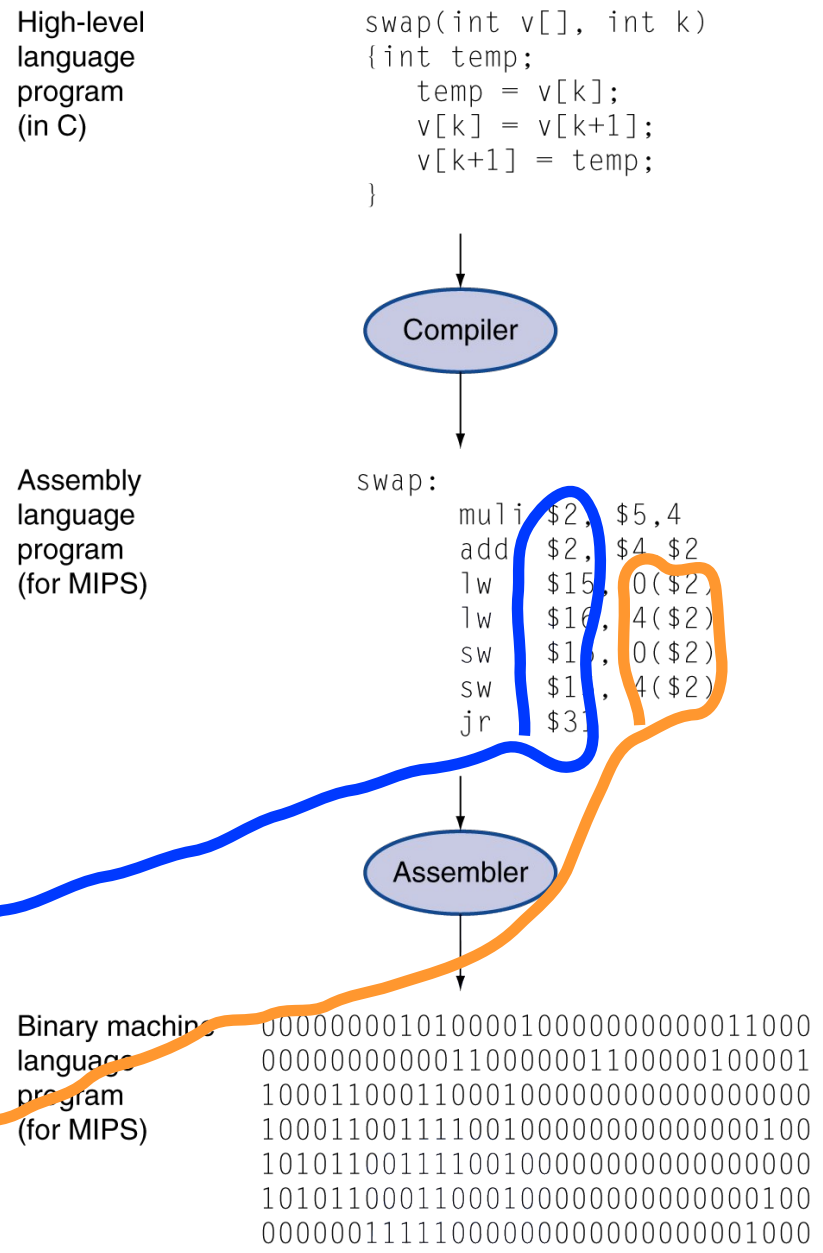
Sections 2.1-2.3

Instruction Set

- **Def:** the repertoire of instructions of a computer

- Instruction sets perform similar functions
  - *1) Arithmetic & logical operations*
  - *2) Memory and port transfers*
  - *3) Flow control*

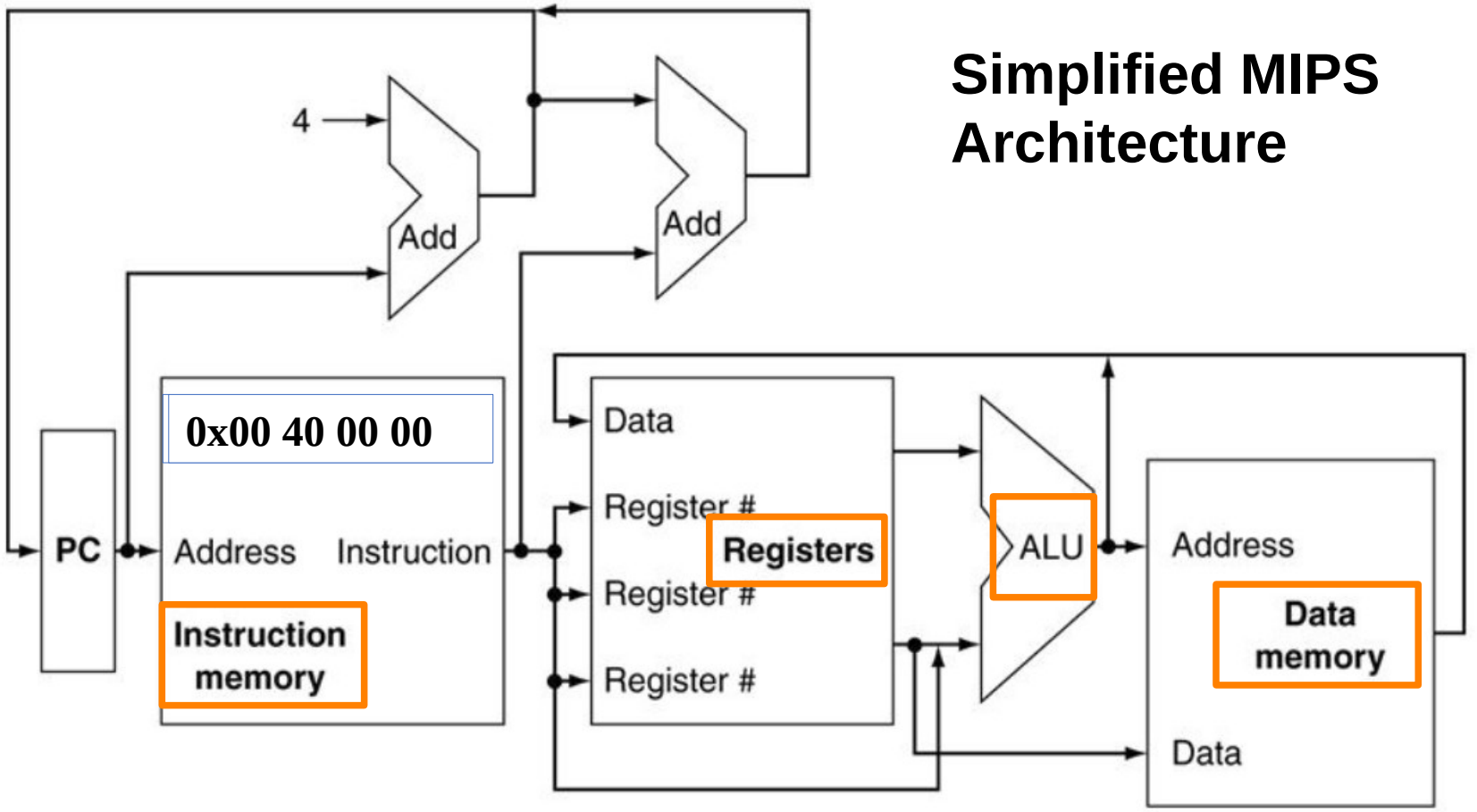
- Each instruction involves **operands** that could be ***processor-specific registers*** and/or ***memory content***



Instruction set design approaches

Basis	RISC	CISC
Full Form	RISC stands for **Reduced Instruction Set Computer**.	CISC stands for **Complex Instruction Set Computer**
Type of Instruction	RISC processors have simple instructions taking about one clock cycle.	CSIC processor has complex instructions that take up multiple clocks for execution.
Instruction Set	The instruction set is reduced i.e. it has only a **few** instructions in the instruction set. Many of these instructions are very **primitive**.	The instruction set has a variety of different instructions that can be used for complex operations.
Execution Time	In RISC Execution time is relatively small.	In CISC Execution time is very high.
Examples	The most common RISC microprocessors are Alpha, ARC, **ARM**, AVR, **MIPS**, PA-RISC, PIC, Power Architecture, and SPARC.	Examples of CISC processors are the System/360, VAX, PDP-11, Motorola 68000 family, **AMD** and **Intel** x86 CPUs.
Average CPI	The average CPI is 1.5 in RISC	The average CPI is in the range of 2 and 15
Focus on	Software Centric Design	Hardware Centric Design

Machine Language Execution



Simplified MIPS Architecture

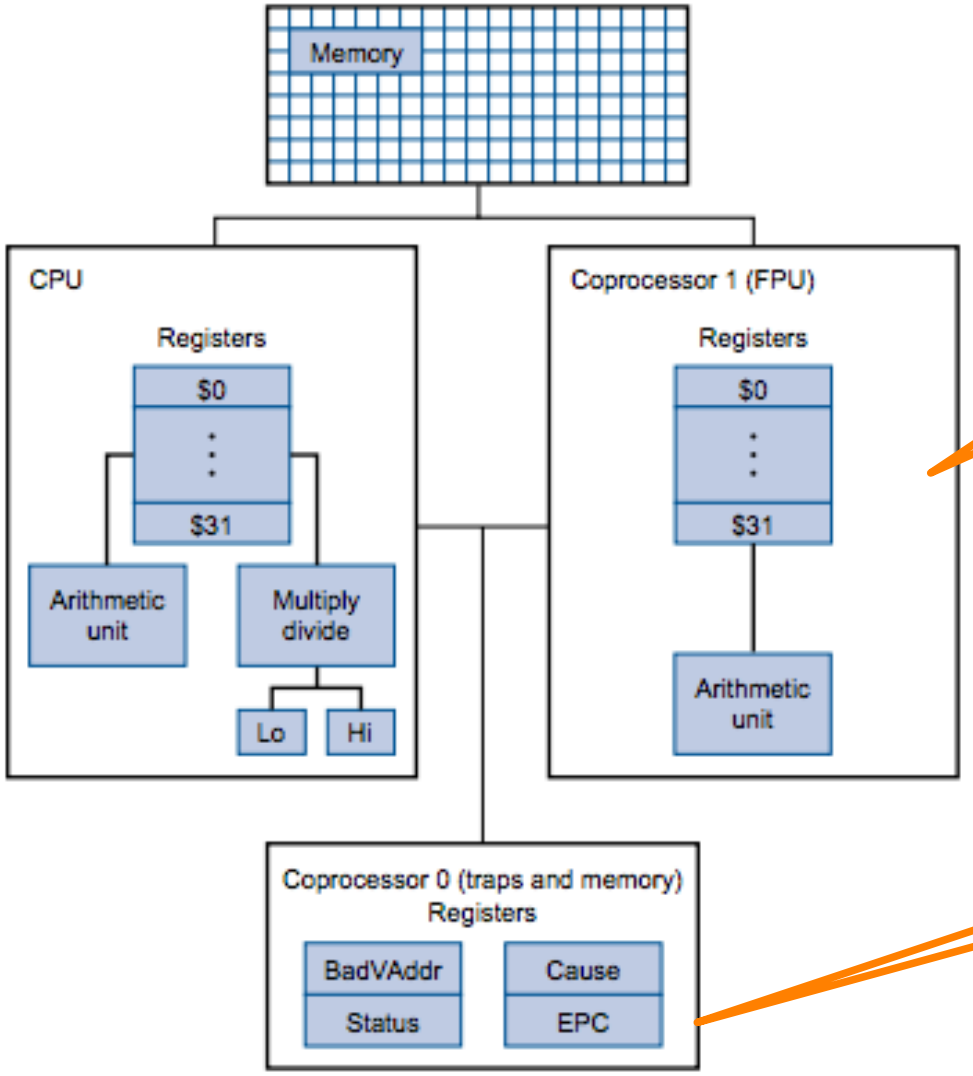
- We will use Stanford MIPS commercialized by MIPS Technologies (www.mips.com)

Objectives

Identify	Identify key instruction types
Understand	Understand machine language execution
Develop	Applications using MIPS assembly

LAB: Thursday 12-1 G24

MIPS Registers



Module 3

Lab

MIPS Main CPU Registers

Name	Register number	Usage
$zero	0	The 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	Saved
$t8–$t9	24–25	More temporaries
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address

32 × ***32-bit*** register file

Registers are used for frequently accessed data

Assembler notation
$a0 → $4

32-bit data → "word"
Default MIPS data unit

MIPS Instruction Set: An Overview

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three register operands
	add immediate	addi $s1,$s2,20	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	lw $s1,20($s2)	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	sw $s1,20($s2)	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	lh $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	lhu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	sh $s1,20($s2)	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	lb $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	lbu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	sb $s1,20($s2)	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	ll $s1,20($s2)	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	sc $s1,20($s2)	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	lui $s1,20	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1,$s2,25	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	slti $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	sltiu $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

MIPS Integer Arithmetic Instructions

Integer Arithmetic

- Processors may have different capabilities: ***data unit*** (e.g., 8-bit, 16-bit, 32-bit,..) and ***types of operands*** (integer only, any numbers)
- MIPS can process both integer and floating-point numbers
  - a **special coprocessor** is used for floating point operations
  - MIPS also has both **32-bit** and 64-bit architectures (but we only focus on 32-bit MIPS architecture)
- Integers (-ve) .... 0 ..... (+ve)
- We consider ***sign-bit representation*** for negative numbers (simpler ALU HW)

	unsigned	Signed
Range	0 to $(2^n - 1)$ 32-bit (0 to +4,294,967,295)	(-2^{n-1}) to $(2^{n-1}-1)$ 32-bit (–2,147,483,648 to +2,147,483,647)

Sections 2.4

2s – Complement Signed Integers

- ***Bit 31*** is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers

- Non-negative numbers have the same unsigned and 2s-complement representation

- Some specific numbers
  - **0**: 0000 0000 … 0000
  - **–1**: 1111 1111 … 1111
    - Most-positive: 0111 1111 … 1111
    - Most-negative: 1000 0000 … 0000

Example

$10_{10} = 00....01010_2$

$-10_{10} = 11....10110_2$

MIPS Arithmetic Operations

- E.g., Add and subtract
- Exactly three ***operands***
  - Two sources and one destination

Assembly instruction operands are either registers or memory operand

a = b + c ***add*** $t0, $S1 $S2 *# t0 = S1 + S2*

MIPS Arithmetic instructions ***ONLY*** use register operands

Key Integer MIPS Instructions

Instruction	RTL	Notes
add $rd, $rs, $rt	R[$rd] ← R[$rs] + R[$rt]	Exception on signed overflow
addu $rd, $rs, $rt	R[$rd] ← R[$rs] + R[$rt]	
sub $rd, $rs, $rt	R[$rd] ← R[$rs] - R[$rt]	Exception on signed overflow
subu $rd, $rs, $rt	R[$rd] ← R[$rs] - R[$rt]	
mtlo $rs	LO ← R[$rs]	
mult $rs, $rt	{HI, LO} ← R[$rs] * R[$rt]	Signed multiplication
multu $rs, $rt	{HI, LO} ← R[$rs] * R[$rt]	Unsigned multiplication
div $rs, $rt	LO ← R[$rs] / R[$rt]HI ← R[$rs] % R[$rt]	Signed division
divu $rs, $rt	LO ← R[$rs] / R[$rt]HI ← R[$rs] % R[$rt]	Unsigned division
mfhi $rd	R[$rd] ← HI	
mflo $rd	R[$rd] ← LO	

Immediate Operands

- ***Constant data*** specified in an instruction

  addi *$s3, $s3, 4*

addi $rt, $rs, imm	R[$rt] ← R[$rs] + SignExt_{16b}(imm)	Exception on signed overflow
addiu $rt, $rs, imm	R[$rt] ← R[$rs] + SignExt_{16b}(imm)	

- *No subtract immediate* instruction
  - Just use a negative constant

    addi *$s2, $s1, -1*

- In MIPS instruction set
  - addi sign extends immediate value

what would be a common use for addi?

The Zero Register

- MIPS register 0 ($zero) is the constant 0
  - Hardwired (Cannot be overwritten)
- Useful for common operations

```
add $t2, $s1, $zero  #no need for move instruction
addi $t2, $zero, 5 #a new load immediate (li) instruct
                   # canbe used for variable initializatic
```

Register Operand Example

- High-level code:

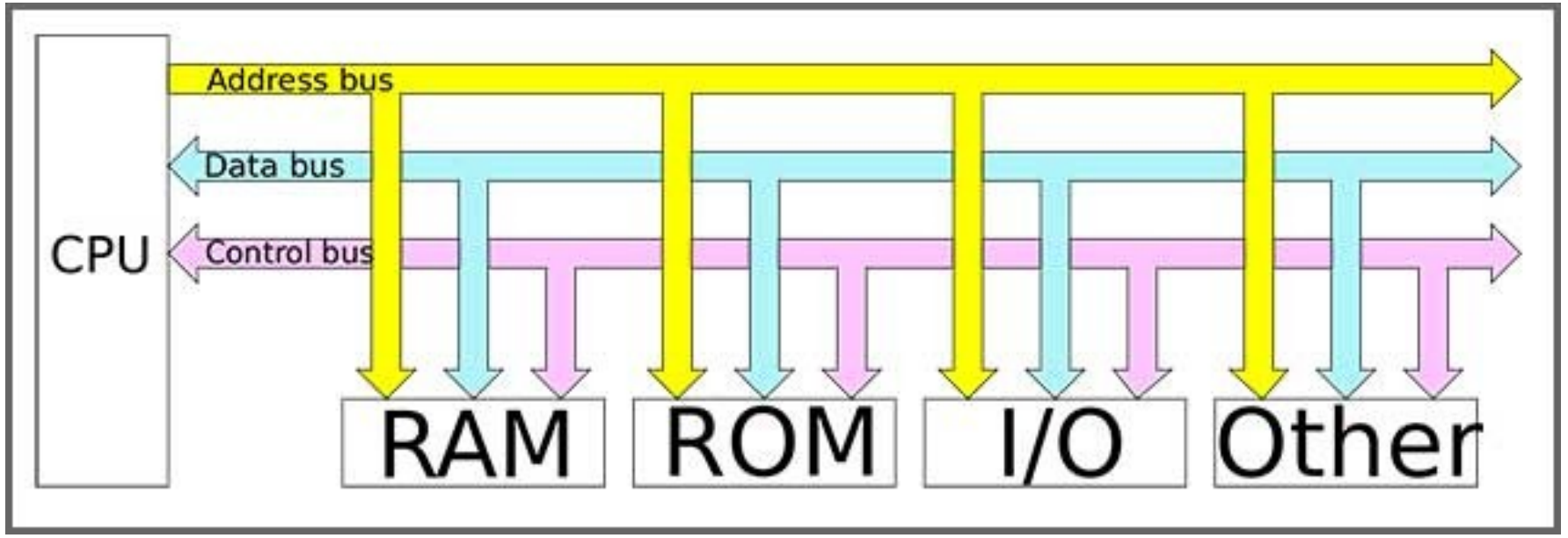
  f = (g + h) - (i + j);
  - **ASSUME** f, …, j in $s0, …, $s4
- Compiled MIPS Assembly:

```
add $t0, $s1, $s2     # t0 = g+h
add $t1, $s3, $s4     # t1 = i+j
sub $s0, $t0, $t1     # s0 = t0 – t1
```

MIPS (similar to all processors) has a limited number of registers.

What to do with programs that process a large number of variables?

Memory related instructions
(read from and write to memory)



CPU puts the memory location address on the address bus

CPU activates control line to indicate whether it is a read or write operation

CPU puts (reads) data on (from) the data bus from (to) one of its registers

Register vs. Memory Performance

- Registers are faster to access than memory
- Operating on memory data requires **load** and **store**
  - More instructions to be executed
- Programmer [Compiler] must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data

Data transfer instructions

Load values from memory into registers

Store result from register to memory

MIPS Memory is ***byte-addressed***

→ you can read/write byte or larger data unit

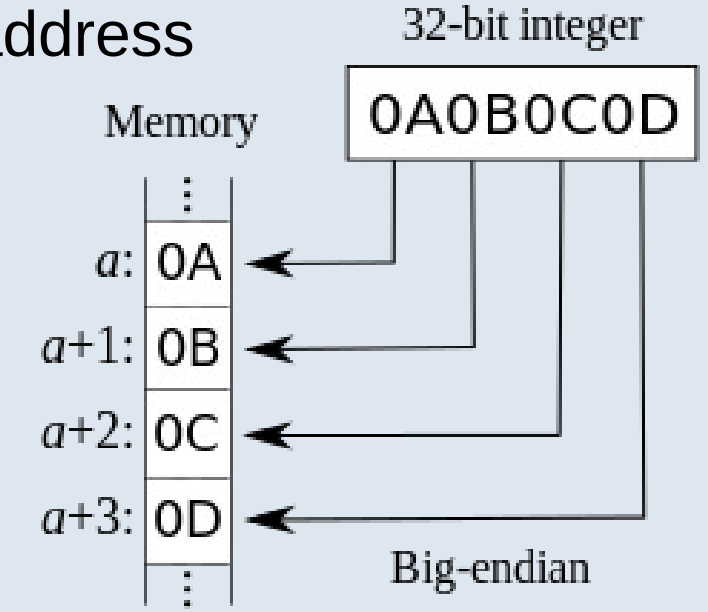
MIPS words are ***aligned*** in memory

- Word **can be only** accessible at an address that is a multiple of 4.
- Half-word address @ multiple of 2

MIPS is Big Endian

Most-significant byte at least address of a word

c.f. Little Endian: least-significant byte at least address



Memory Operand Example 1

- C code:

  g = h + A[8];

  A[] is an integer array

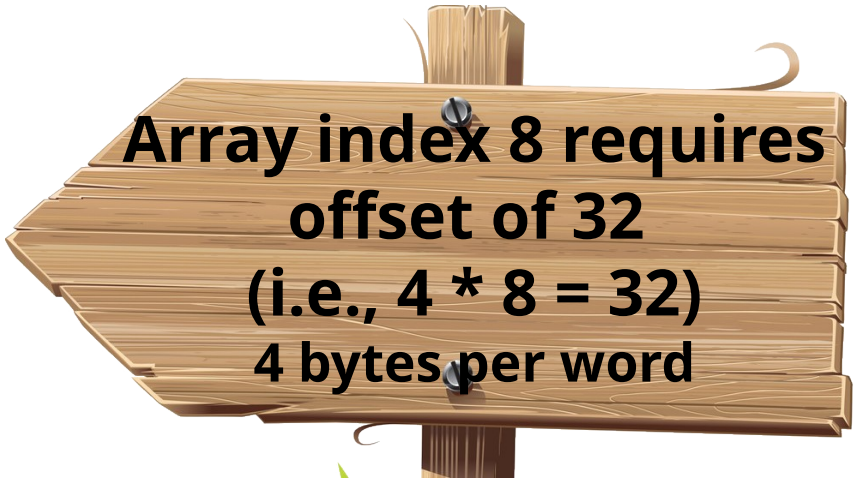
  - g in $s1, h in $s2, base address of A in $s3

- Compiled MIPS code:

```
lw  $t0, 32($s3)    # load word
add $s1, $s2, $t0
```

offset: byte offset from the array start

base register: *address of first item in the array*



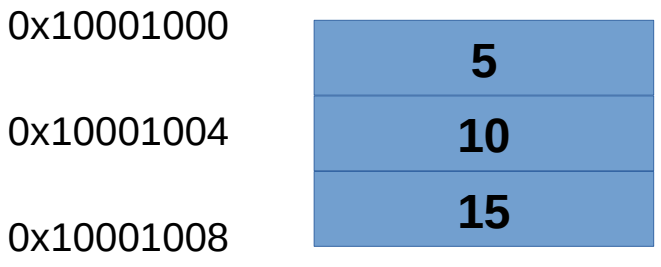
A program that loads an integer array of three elements from the memory ***without*** using loop and print the sum

```
3  .data
4  array: .word 5, 10, 15 # Integer array with three elements
5  .text
6  main:
7      # Load the three array elements into registers
8      lw $t0, array          # Load the first element into $t0
9      lw $t1, array+4        # Load the second element into $t1
10     lw $t2, array+8        # Load the third element into $t2
11     # Calculate the sum of the array elements
12     add $t3, $t0, $t1      # Add the first and second elements
13     add $t3, $t3, $t2      # Add the third element to the sum
14     # Print the sum
15     move $a0, $t3          # Move the sum to $a0 for printing
16     li $v0, 1              # Load the print integer syscall code
17     syscall
18     # Exit the program
19     li $v0, 10             # Load the exit syscall code
20     syscall
```

```
1  .data
2  array:      .word 5, 10, 15  # Integer array with three elements
3  .text
4  main:
5      la $t4, array    # Load the base address of the array into $t4
6      lw $t0, 0($t4)     # Load the first element into $t0
7      lw $t1, 4($t4)     # Load the second element into $t1
8      lw $t2, 8($t4)     # Load the third element into $t2
9      # Calculate the sum of the array elements
10     add $t3, $t0, $t1      # Add the first and second elements
11     add $t3, $t3, $t2      # Add the third element to the sum
12     # Print the sum
13     move $a0, $t3          # Move the sum to $a0 for printing
14     li $v0, 1              # Load the print integer syscall code
15     syscall
16     # Exit the program
17     li $v0, 10             # Load the exit syscall code
18     syscall
```

Same program using standard lw instruction

*lw $t4, **OFFSET**(**base**)*



La $t4, array → loads address instruction of array label → $t4=0x10001000
Lw $t4, array → loads 4 bytes starting at array label → $t4=5
li $t4, 25 → loads $t4 with the provided immediate value → $t4=25

Key Memory MIPS Instructions

Instruction	RTL	Notes
lb $rt, imm($rs)	$R[\$rt] \leftarrow SignExt_{8b}(Mem_{1B}(R[\$rs] + SignExt_{16b}(imm)))$	
lh $rt, imm($rs)	$R[\$rt] \leftarrow SignExt_{16b}(Mem_{2B}(R[\$rs] + SignExt_{16b}(imm)))$	Computed address must be a multiple of 2
lw $rt, imm($rs)	$R[\$rt] \leftarrow Mem_{4B}(R[\$rs] + SignExt_{16b}(imm))$	Computed address must be a multiple of 4
lbu $rt, imm($rs)	$R[\$rt] \leftarrow \{0 \times 24, Mem_{1B}(R[\$rs] + SignExt_{16b}(imm))\}$	
lhu $rt, imm($rs)	$R[\$rt] \leftarrow \{0 \times 16, Mem_{2B}(R[\$rs] + SignExt_{16b}(imm))\}$	Computed address must be a multiple of 2
sb $rt, imm($rs)	$Mem_{1B}(R[\$rs] + SignExt_{16b}(imm)) \leftarrow (R[\$rt])[7:0]$	
sh $rt, imm($rs)	$Mem_{2B}(R[\$rs] + SignExt_{16b}(imm)) \leftarrow (R[\$rt])[15:0]$	Computed address must be a multiple of 2
sw $rt, imm($rs)	$Mem_{4B}(R[\$rs] + SignExt_{16b}(imm)) \leftarrow R[\$rt]$	Computed address must be a multiple of 4

Signed Extension

- Needed when loading a byte, or half-word to MIPS 32-bit registers
  - **unsigned values:** extend with 0s
  - **Signed numbers:** replicate the sign bit to the left
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110
- In MIPS instruction set
  - lb, lh: extend loaded byte/half-word in the register

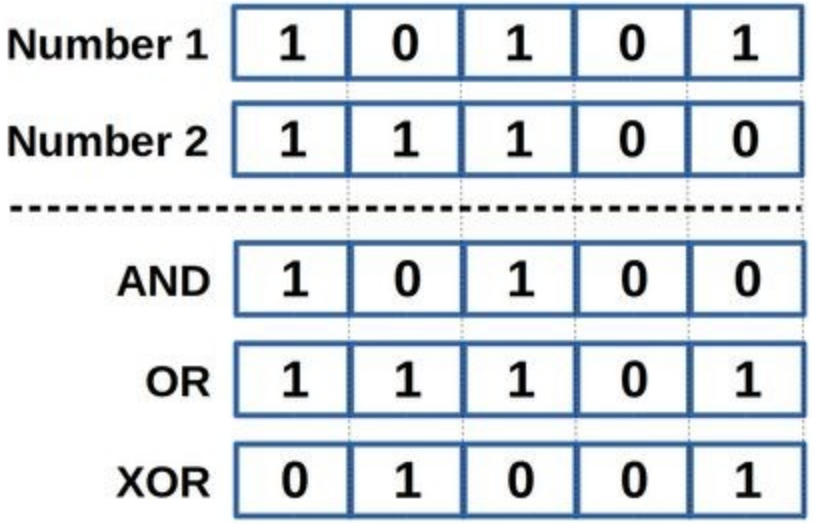
MIPS Logic Instructions

Logical Operations

Sections 2.6

- Instructions for ***bitwise*** manipulation

Operation	C	Java	MIPS
Shift left	<<	<<	***sll***
Shift right	>>	>>>	***srl***
Bitwise AND	&	&	***and, andi***
Bitwise OR	|	|	***or, ori***
Bitwise NOT	~	~	***nor***



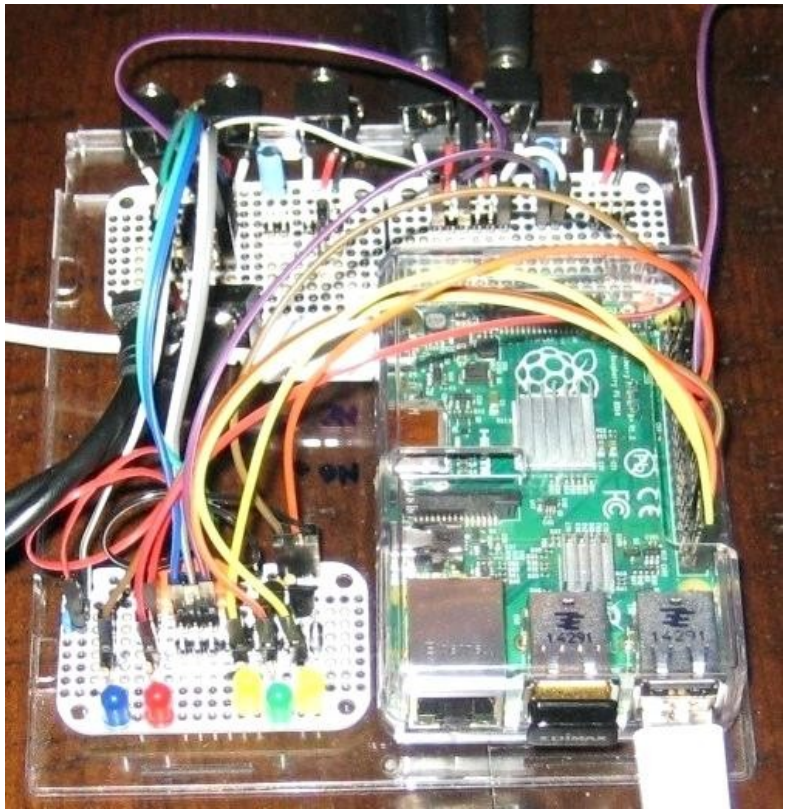
Logical Instructions (All are R-format)

Instruction	*RTL*	*Notes*
sll $rd, $rt, shamt	*R[$rd] ← R[$rt] << shamt*	***sll*** by *i* bits multiplies by 2^i
srl $rd, $rt, shamt	*R[$rd] ← R[$rt] >> shamt*	***srl*** by *i* bits divides by 2^i (unsigned)
sra $rd, $rt, shamt	*R[$rd] ← R[$rt] >> shamt*	*Signed right shift*
sllv $rd, $rt, $rs	*R[$rd] ← R[$rt] << R[$rs]*	
srlv $rd, $rt, $rs	*R[$rd] ← R[$rt] >> R[$rs]*	*Unsigned right shift*
srav $rd, $rt, $rs	*R[$rd] ← R[$rt] >> R[$rs]*	*Signed right shift*
and $rd, $rs, $rt	*R[$rd] ← R[$rs] & R[$rt]*	
or $rd, $rs, $rt	*R[$rd] ← R[$rs] | R[$rt]*	
xor $rd, $rs, $rt	*R[$rd] ← R[$rs] ^ R[$rt]*	
nor $rd, $rs, $rt	*R[$rd] ← !(R[$rs] | R[$rt])*	

Relevant Applications

Logical operators enables the programmer (or the compiler) to test and set specific bits connected to computer ports

bits could have a physical meaning
- Alarm ON/OFF (Input)
- Status led (ON/OFF) [output]



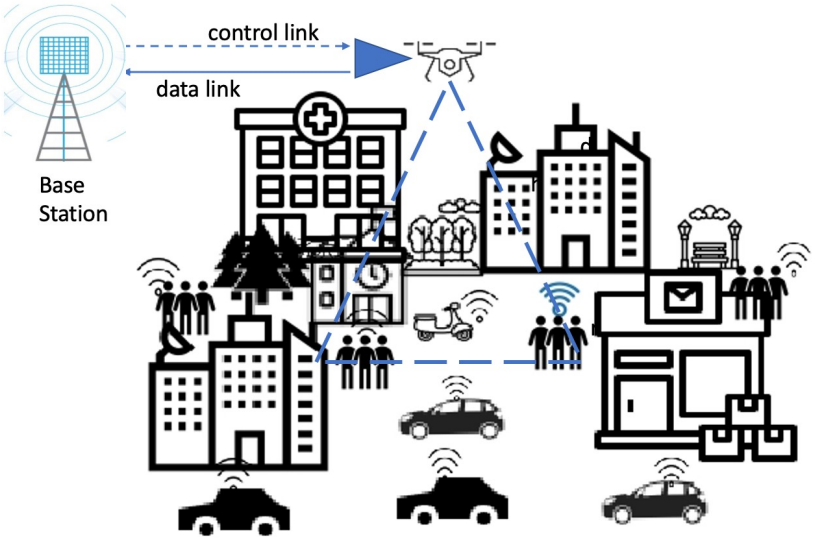
Efficient program development?!!!

Efficient program development?

Bitmaps instead of an integer array!

64 resource → used or not?

The choice of data unit makes a big impact on computing and communication.



AND Operations

- Useful to **checking (testing)** the value of ***specific*** bits in a word (register)
  - Is a button pressed?
  - Find the value of a sensor from multiple bits

For AND operation:
The mask has “1”s at test bit locations and “0”s otherwise.

STEPS:

1- The port is read to register $t2
2- Mask is defined (loaded) in mask register $t1
3- AND mask and data registers → **$t0**
4- test the outcome and take actions

$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

AND Example

```
4   .text
5   main:
6       lw $t0, 0xFFFFFFF0 # Read the I/O register (at address 0xFFFFFFF0)
7       lw $t1, 0x7800      # Mask to extract bits 11-14
8       and $t1, $t0,$t1   # test by MASK and register
9       srl $t1, $t1, 10   # Shift the result to the right by 10 bits
10      move $a0, $t1    # Move the value of $t1 to $a0 for printing
11      li $v0, 1        # Load the print integer syscall code
12      syscall        # Print the integer reading
13      li $v0, 10   # Load the exit syscall code
14      syscall          # Exit the program
```

NOTE: MIPS use same memory instructions to read I/O ports (RISC instruction set)

OR Operation

- Useful to **set** some bits to 1 in a word and leave others unchanged
  - e.g., switch a LED on, set on a HW switch, set a configuration bit for a port

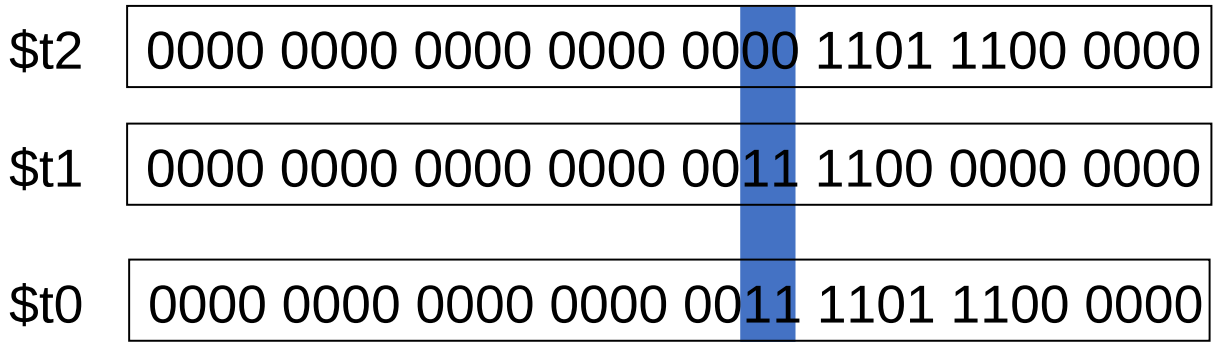
STEPS:

1- The port is read to register $t2

2- relevant bits are set in another register ($t1)

3- OR two registers

4- send the result to the target port



NOT Operations

- Useful to **invert** bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
  - a NOR b == ***NOT*** (a OR b)

NOR *$t0, $t1, $zero* *(remember: A or Zero = A)*

$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111

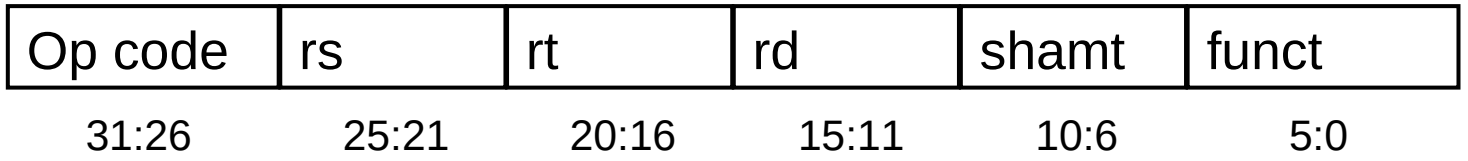
MIPS Instructions Encoding (Assembly → Binary)

MIPS instructions have a fixed size of 32 bits

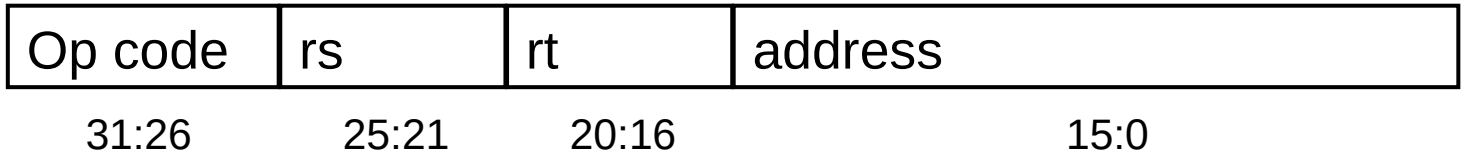
The layout of the 32 bits is defined as the Instruction format

Basic MIPS instructions have three key instruction formats:

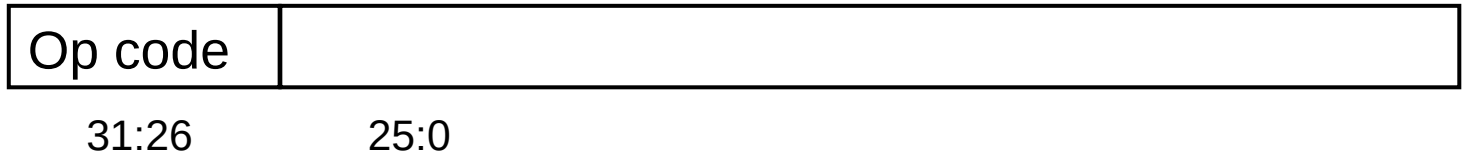
1) R- format (ALU, …)



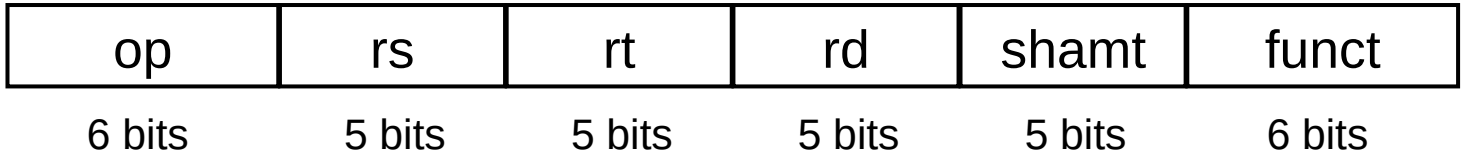
2) I-Format (lw, sw, addi)



3) J-Format (j)



MIPS R-format Instructions



- Fields for Register instruction format
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount (00000 for now)
  - funct: function code (extends opcode)
- ALU instructions, others later

Remember that MIPS has 32 registers ($32 = 2^5$) → 5 bits are needed to identify every register in the register file

R-format Example

High-level language program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

Compiler

Assembly language program (for MIPS)

```
swap:
      muli $2, $5,4
      add  $2, $4,$2
      lw   $15, 0($2)
      lw   $16, 4($2)
      sw   $16, 0($2)
      sw   $15, 4($2)
      jr   $31
```

Assembler

Binary machine language program (for MIPS)

```
00000000101000010000000000011000
00000000000011000000110000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

add *$t0, $s1, $s2*

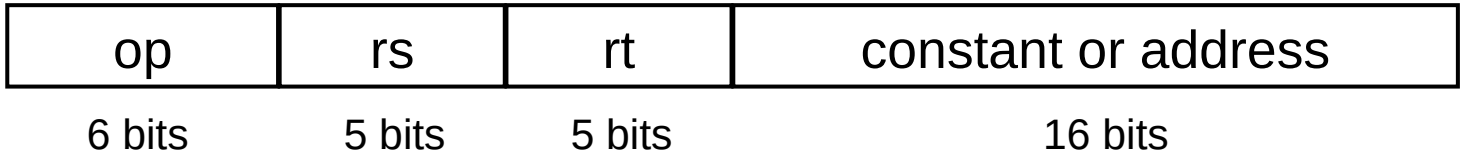
special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

← Decimal

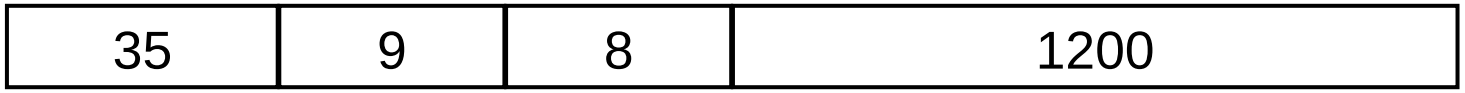
← Binary

$$00000010001100100100000000100000_2 = 02324020_{16}$$

MIPS I-format Instructions



- Immediate arithmetic and load/store instructions
  - rt: data target [**destination** or source] register
  - Constant: -2^{15} to $+2^{15} - 1$
  - Address: offset added to **base address in rs**
- Example: *lw $t0, 1200($t1)*



Signed Extension (revisited)

- Signed extension is also used for extending
  - immediate values (e.g., addi,
  - OFFSET (16 bits) of lw, sw, …

Immediate value is limited by 16 bits.

How does MIPS deal with large numbers?

Supporting Large Constants

Sections 2.10

- Most constants are small
  - 16-bit immediate is sufficient (***make common case fast***)

- For the occasional 32-bit constant (***large constant***) [2 steps]

1- *lui $at, constant*

- Copies Most significant 16-bit constant to left 16 bits of ***$at***
- Clears right 16 bits of $at to 0
- $at (register #1): **assembler temporary**

2- ori lower half

Example: 0000 0000 0111 1101 0000 1001 0000 0000

lui $at, 61

0000 0000 0111 1101	0000 0000 0000 0000

ori $t0, $at, 2304

0000 0000 0111 1101	0000 1001 0000 0000