SoC Design : Lecture 3

Prof. Jong-Myon Kim

울산대학교
UNIVERSITY OF ULSAN

Contents

- Introduction to Instruction Set Architecture

MIPS Instruction Set Architectures

Instruction

- Language of the Machine
- More primitive than higher level languages
  - e.g., no sophisticated control flow
- Very restrictive
  - e.g., MIPS Arithmetic Instructions

- We'll be working with the MIPS instruction set architecture
  - A representative of Reduced Instruction Set Computer (RISC)
  - Similar to other architectures developed since the 1980's
  - Used by NEC, Nintendo, Silicon Graphics, Sony…

Design goals: Maximize performance and Minimize cost, Reduce design time

MIPS arithmetic

- All instructions have 3 operands
- Operand order is fixed (destination first)

Example:

C code: `A = B + C`

MIPS code: `add $s0, $s1, $s2`

(associated with variables by compiler)

MIPS arithmetic

- Design Principle: simplicity favors regularity.
- Of course this complicates some things...

C code:

```
A = B + C + D;
E = F - A;
```

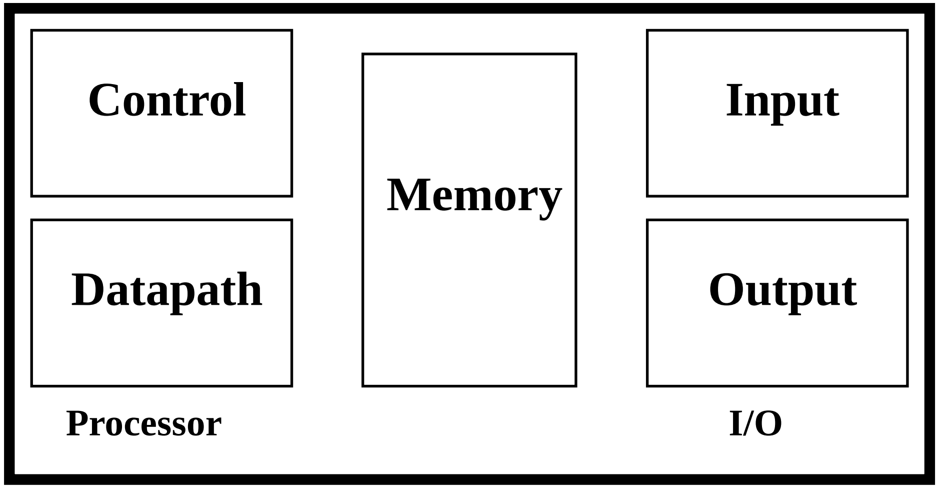
MIPS code:

```
add $t0, $s1, $s2
add $s0, $t0, $s3
sub $s4, $s5, $s0
```

- Operands must be registers, only 32 registers provided
- Each register contains 32 bits
- All memory accesses are accomplished via loads and stores
  - A common feature of RISC processors

Registers vs. Memory

- Arithmetic instructions operands must be registers,
  — only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables



Memory Organization

- Viewed as a large, single-dimension array, with an address
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory

0	**8 bits of data**
1	**8 bits of data**
2	**8 bits of data**
3	**8 bits of data**
4	**8 bits of data**
5	**8 bits of data**
6	**8 bits of data**
...	

Memory Organization

- Bytes are nice, but most data items use larger "words"
- MIPS provides lw/lh/lb and sw/sh/sb instructions
- For MIPS, a word is 32 bits or 4 bytes.
  - (Intel's word=16 bits and double word or dword=32bits)

0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data

...

Registers hold 32 bits of data

- 2^{32} bytes with byte addresses from 0 to 2^{32}-1
- 2^{30} words with byte addresses 0, 4, 8, ... 2^{32}-4
- Words are aligned

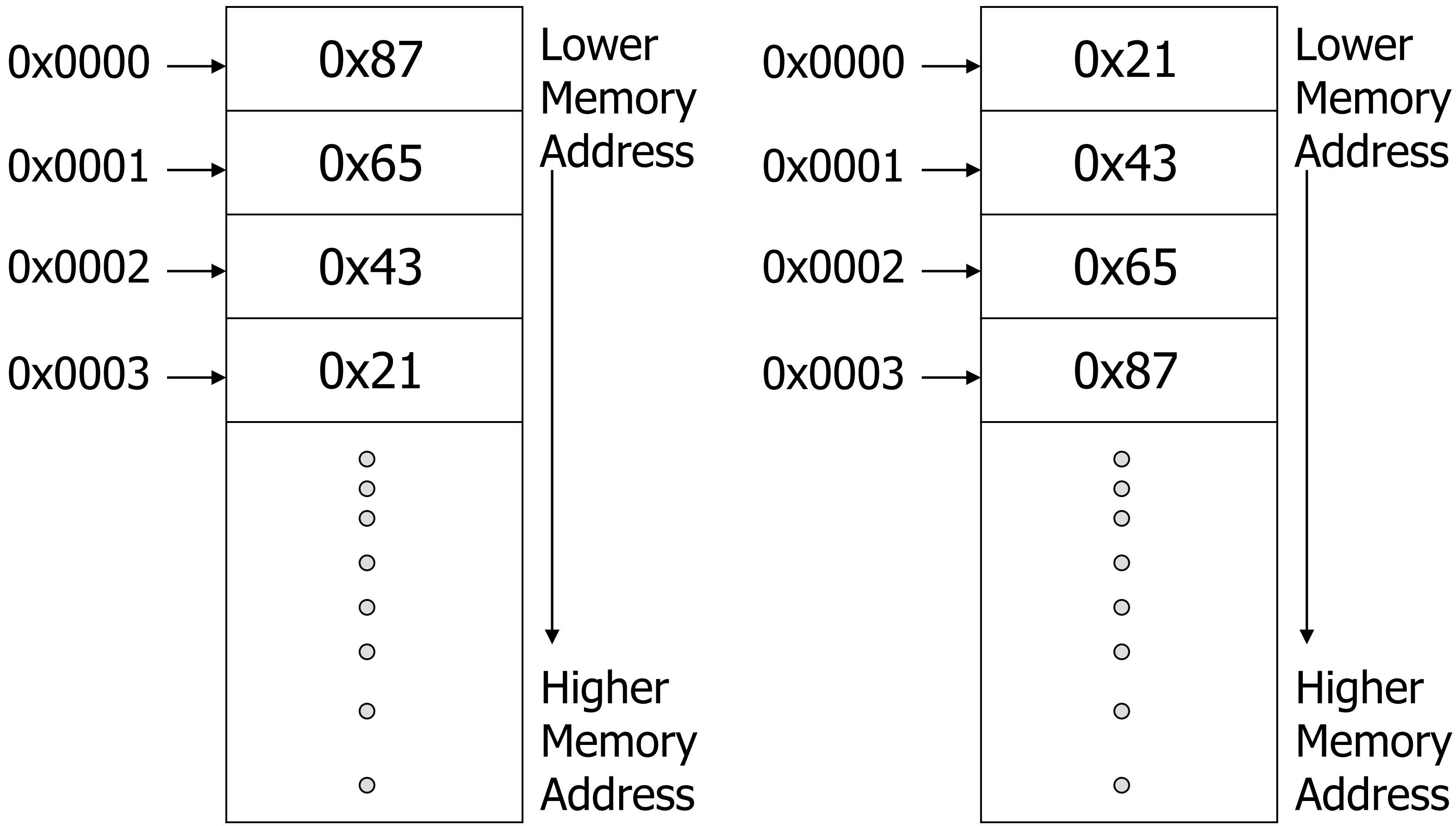
  i.e., what are the least 2 significant bits of a word address?

Endianness [defined by Danny Cohen 1981]

- Byte ordering — How a multiple byte data word stored in memory
- Endianness (from Gulliver's Travels)
  - Big Endian
    - Most significant byte of a multi-byte word is stored at the lowest memory address
    - e.g. Sun Sparc, PowerPC
  - Little Endian
    - Least significant byte of a multi-byte word is stored at the lowest memory address
    - e.g. Intel x86
- Some embedded & DSP processors would support both for interoperability

Example of Endian

- Store 0x87654321 at address 0x0000, byte-addressable



BIG ENDIAN **LITTLE ENDIAN**

Instructions

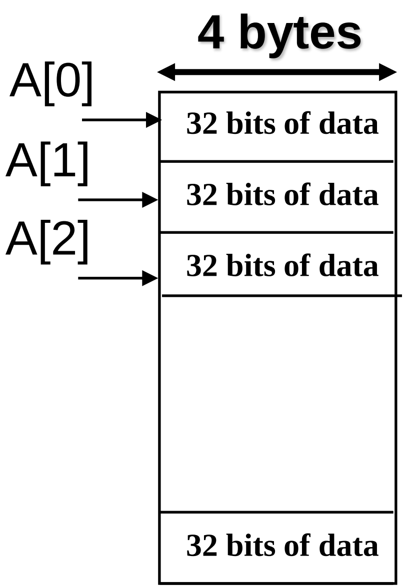
- Load and store instructions
- Example:

C code:

```
long A[100];
A[9] = h + A[8];
```

MIPS code:

```
lw  $t0, 32($s3)
add $t0, $s2, $t0
sw  $t0, 36($s3)
```



- Store word has destination last
- Remember arithmetic operands are registers, not memory!

Can't write: add 48($s3), $s2, 32($s3)

Our First Example

- Can we figure out the code?

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

```
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
```

- MIPS Software Convention
  - $4, $5, $6, $7 are used for passing arguments

So far we've learned:

- MIPS
  - — loading words but addressing bytes
  - — arithmetic on registers only

- Instruction / Meaning

Instruction	Meaning
`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`
`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`
`lw $s1, 100($s2)`	`$s1 = Memory[$s2+100]`
`sw $s1, 100($s2)`	`Memory[$s2+100] = $s1`

Software Conventions for MIPS Registers

Register	Names	Usage by Software Convention
$0	$zero	Hardwired to zero
$1	$at	Reserved by assembler
$2 - $3	$v0 - $v1	Function return result registers
$4 - $7	$a0 - $a3	Function passing argument value registers
$8 - $15	$t0 - $t7	Temporary registers, caller saved
$16 - $23	$s0 - $s7	Saved registers, callee saved
$24 - $25	$t8 - $t9	Temporary registers, caller saved
$26 - $27	$k0 - $k1	Reserved for OS kernel
$28	$gp	Global pointer
$29	$sp	Stack pointer
$30	$fp	Frame pointer
$31	$ra	Return address (pushed by call instruction)
$hi	$hi	High result register (remainder/div, high word/mult)
$lo	$lo	Low result register (quotient/div, low word/mult)

Instruction Format

- Instruction / Meaning

Instruction	Meaning
`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`
`sub $s1,$s2,$s3`	`$s1 = $s2 – $s3`
`lw $s1,100($s2)`	`$s1 = Memory[$s2+100]`
`sw $s1,100($s2)`	`Memory[$s2+100] = $s1`
`bne $s4,$s5,Label`	`Next instr. is at Label if $s4 ≠ $s5`
`beq $s4,$s5,Label`	`Next instr. is at Label if $s4 = $s5`
`j Label`	`Next instr. is at Label`

- Formats:

R	**op**	**rs**	**rt**	**rd**	**shamt**	**funct**
I	**op**	**rs**	**rt**	**16 bit address**		
J	**op**	**26 bit address**				

Machine Language

- Instructions, like registers and words of data, are also 32 bits long
  - Example: `add $t0, $s1, $s2`
  - registers have numbers, `$t0=9, $s1=17, $s2=18`

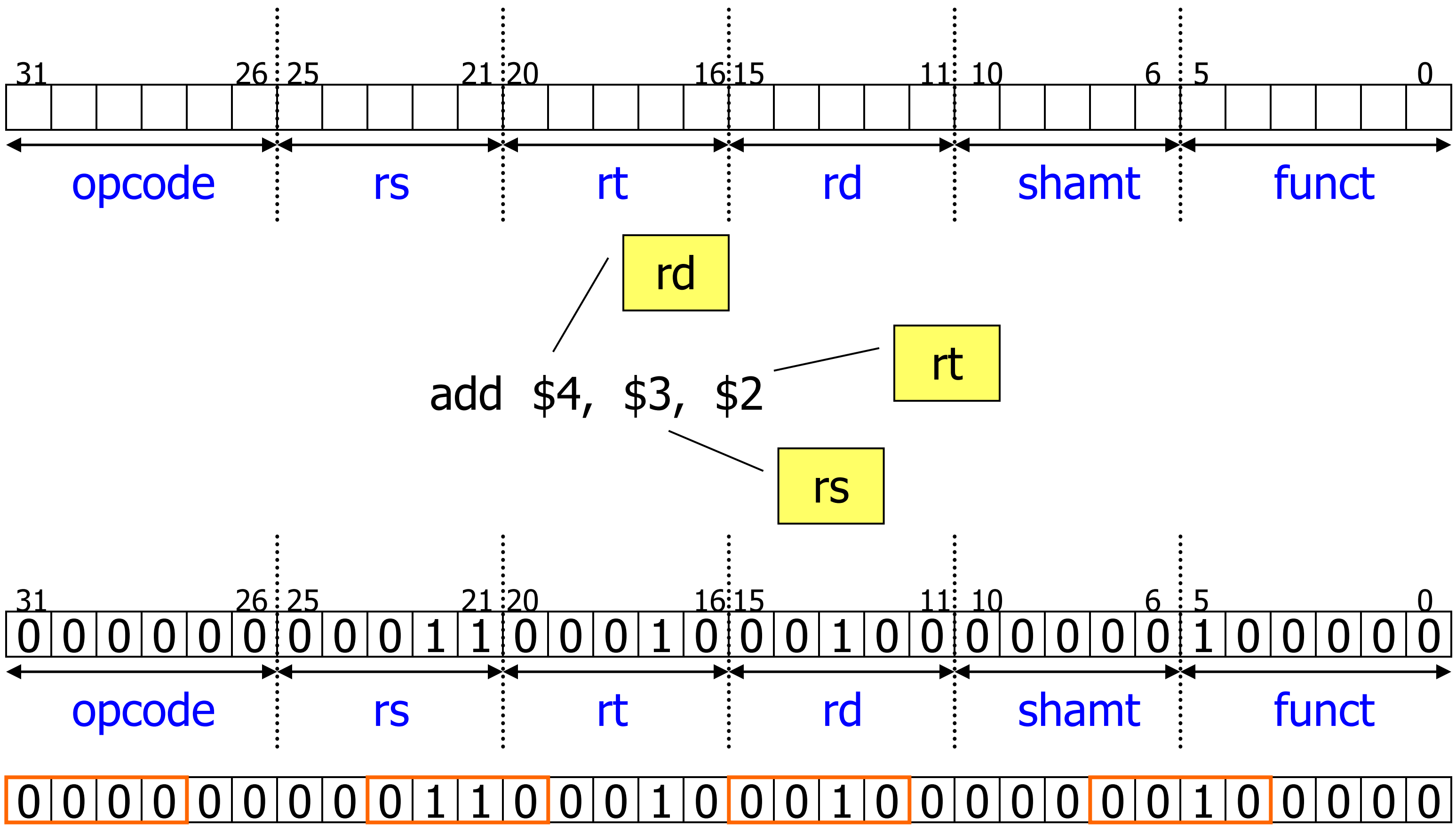
- Instruction Format:

000000	10001	10010	01001	00000	100000

op	rs	rt	rd	shamt	funct

- *Can you guess what the field names stand for?*

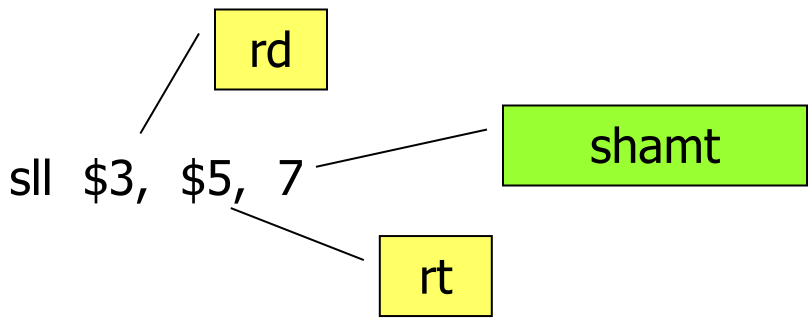
MIPS Encoding: R-Type



Encoding = 0x00622020

MIPS Encoding: R-Type

31–26	25–21	20–16	15–11	10–6	5–0
opcode	rs	rt	rd	shamt	funct



31–26	25–21	20–16	15–11	10–6	5–0
0 0 0 0 0 0	0 0 0 0 0	0 0 1 0 1	0 0 0 1 1	0 0 1 1 1	0 0 0 0 0 0
opcode	rs	rt	rd	shamt	funct

0 0 0 0	0 0 0 0	0 0 0 0	0 1 0 1	0 0 0 1	1 0 0 1	1 1 0 0	0 0 0 0

Encoding = 0x000519C0

Machine Language

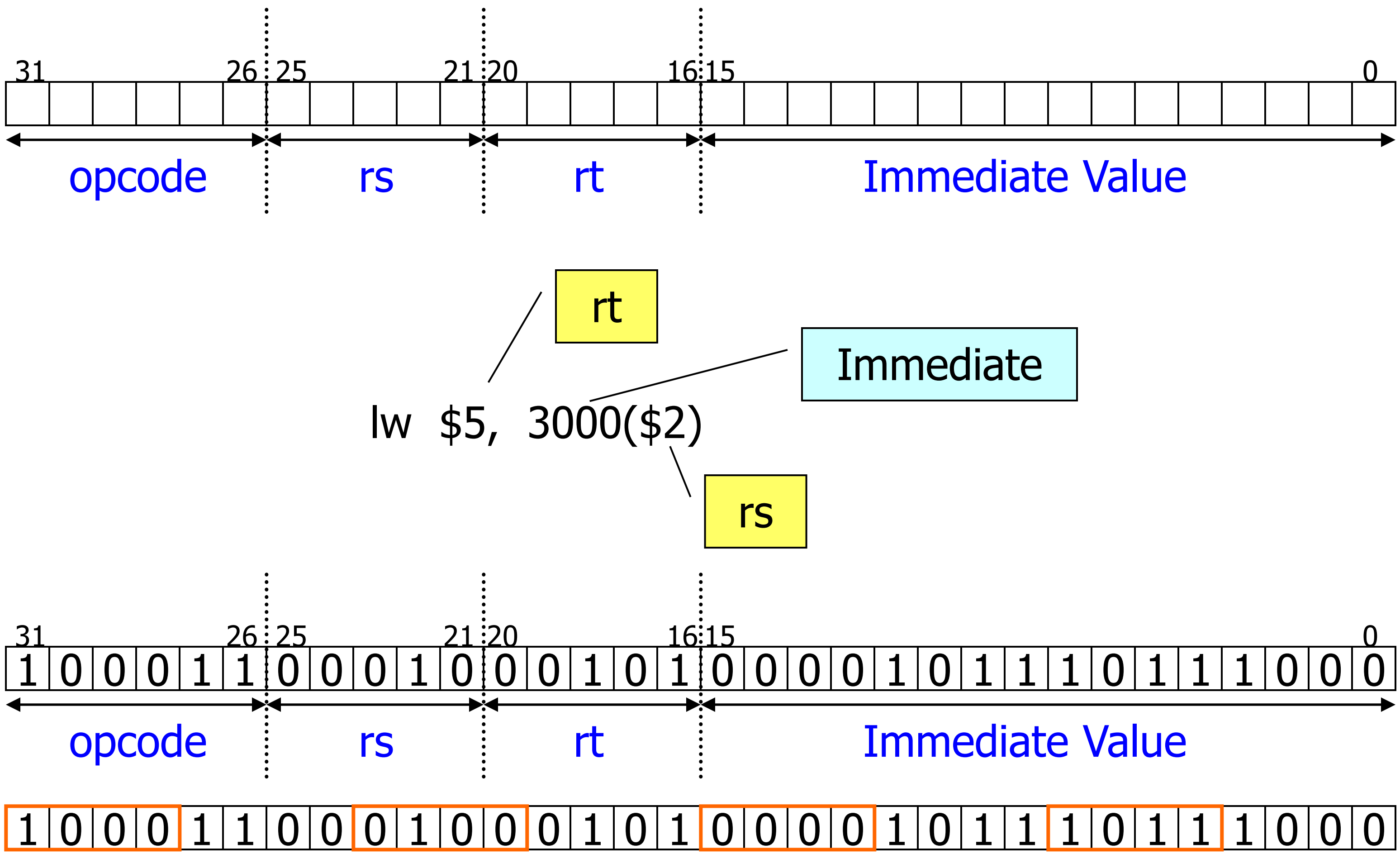
- Consider the load-word and store-word instructions,
  - What would the regularity principle have us do?
  - New principle: Good design demands a compromise
- Introduce a new type of instruction format
  - I-type for data transfer instructions
  - other format was R-type for register
- Example: `lw $t0, 32($s2)`

35	18	9	32

op	rs	rt	16 bit number

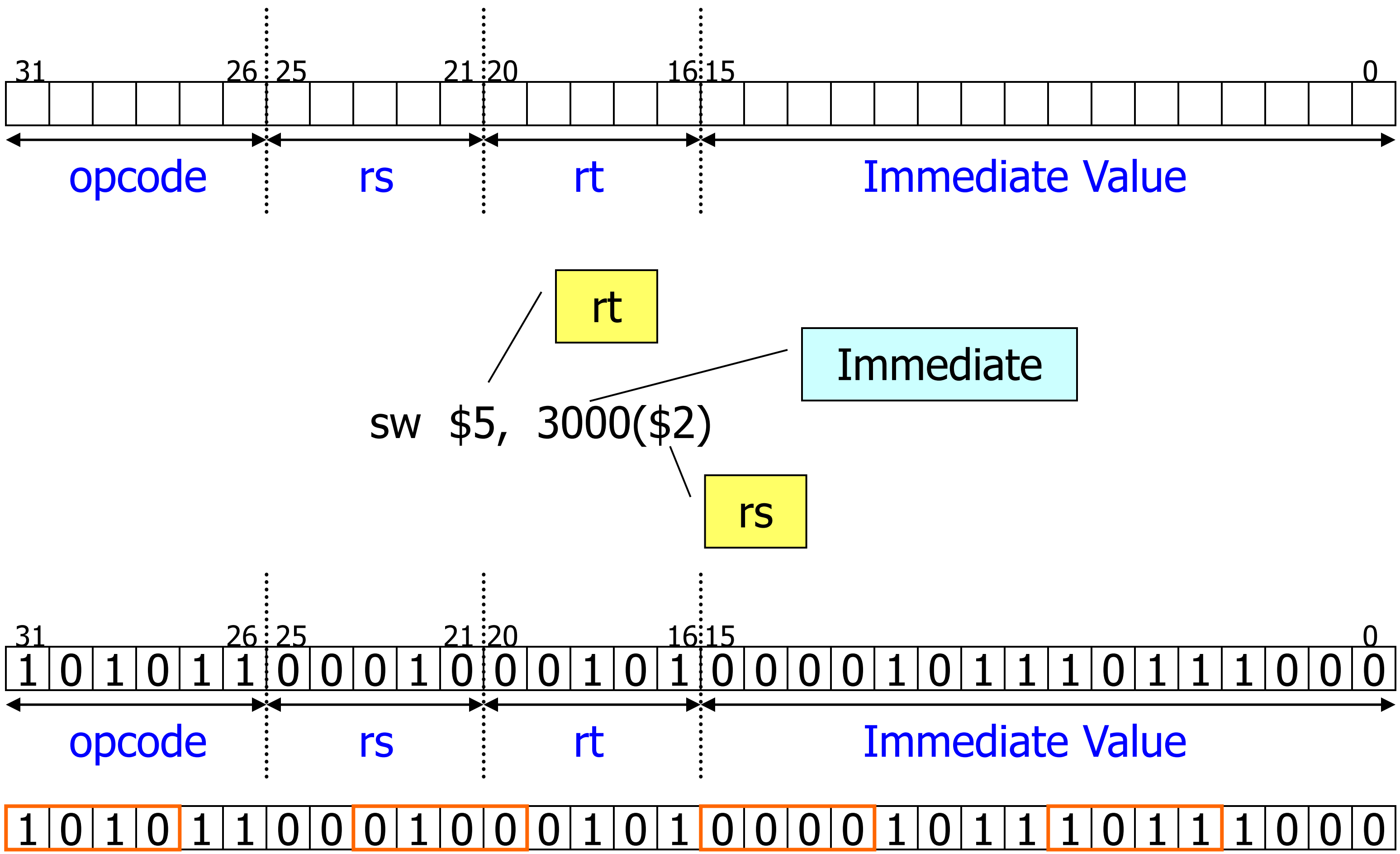
- Where's the compromise?

MIPS Encoding: I-Type



Encoding = 0x8C450BB8

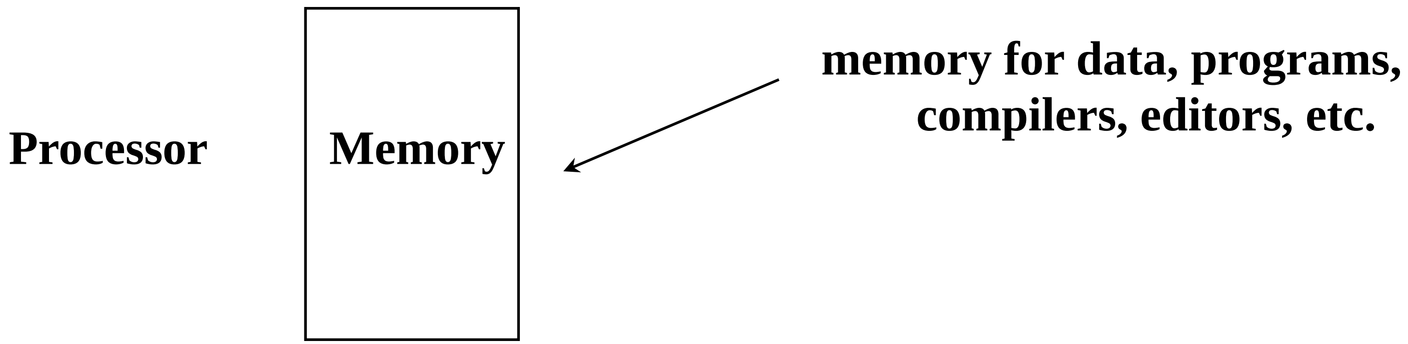
MIPS Encoding: I-Type



Encoding = 0xAC450BB8

Stored Program Concept

- Instructions are bits
- Programs are stored in memory
  — to be read or written just like data



- Fetch & Execute Cycle
  - Instructions are fetched and put into a special register
  - Bits in the register "control" the subsequent actions
  - Fetch the “next” instruction and continue

Control

- Decision making instructions
  - alter the control flow,
  - i.e., change the "next" instruction to be executed

- MIPS conditional branch instructions:

```
bne $t0, $t1, Label
beq $t0, $t1, Label
```

- Example: if (i==j) h = i + j;

```
          bne $s0, $s1, Label
          add $s3, $s0, $s1
Label: ....
```

Control

- MIPS unconditional branch instructions:

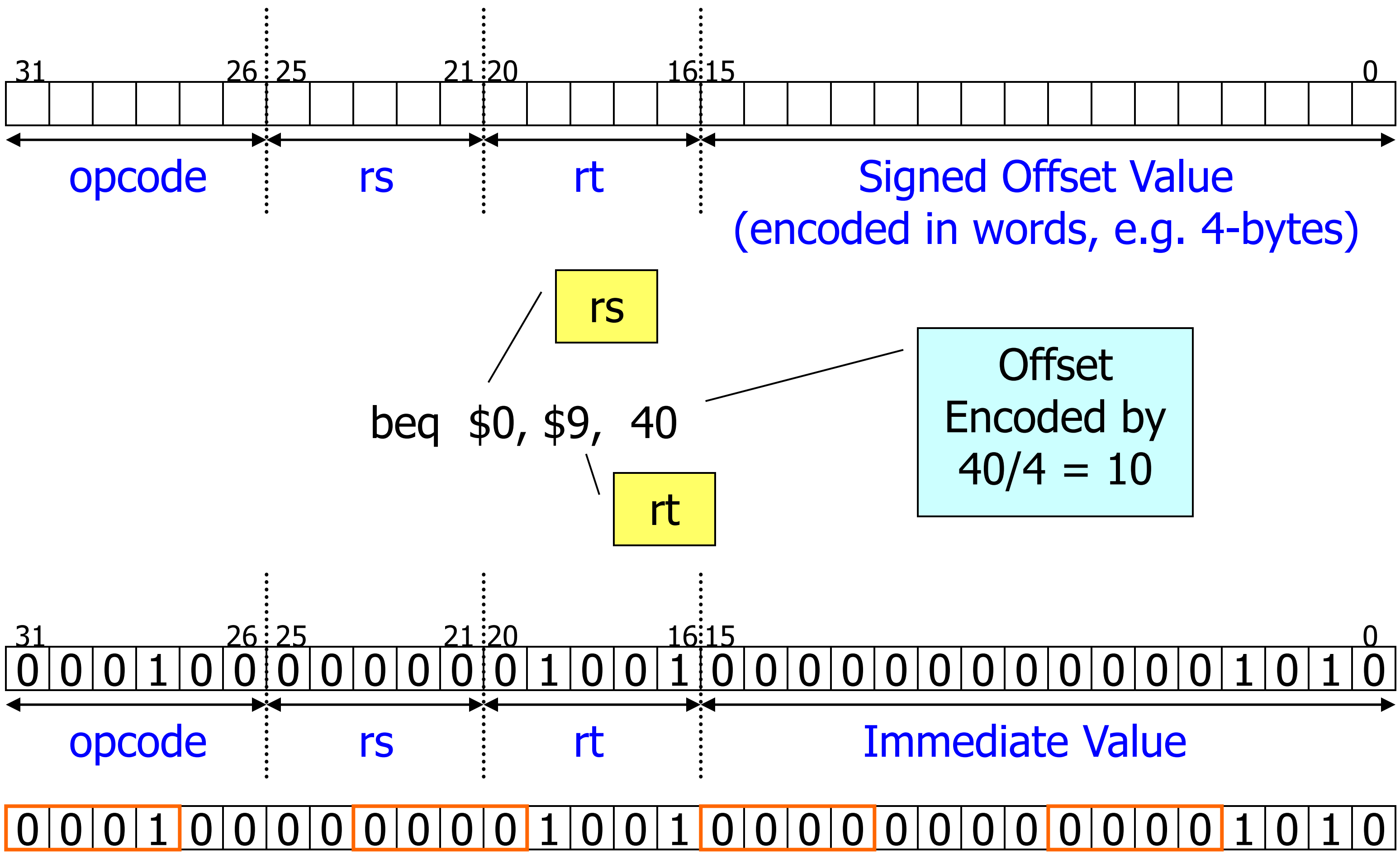
```
j  label
```

- Example:

```
if (i!=j)              beq $s4, $s5, Lab1
    h=i+j;             add $s3, $s4, $s5
else                   j Lab2
    h=i-j;             Lab1:  sub $s3, $s4, $s5
                       Lab2:  ...
```

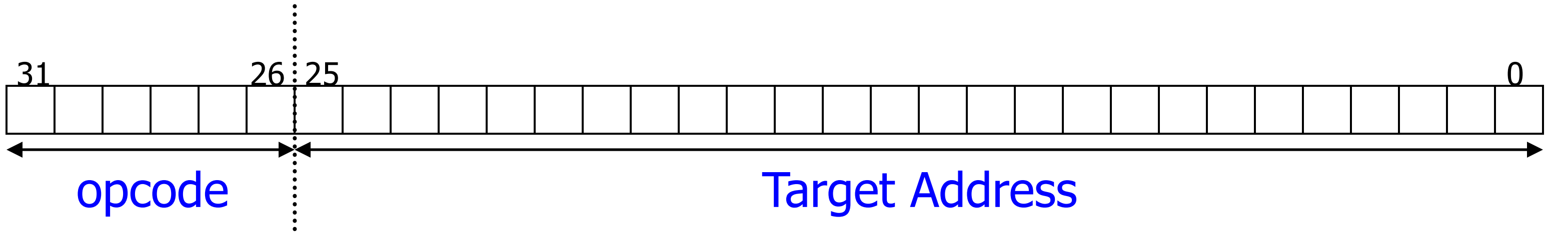
- *Can you build a simple for loop?*

BEQ/BNE uses I-Type



Encoding = 0x1009000A

MIPS Encoding: J-Type



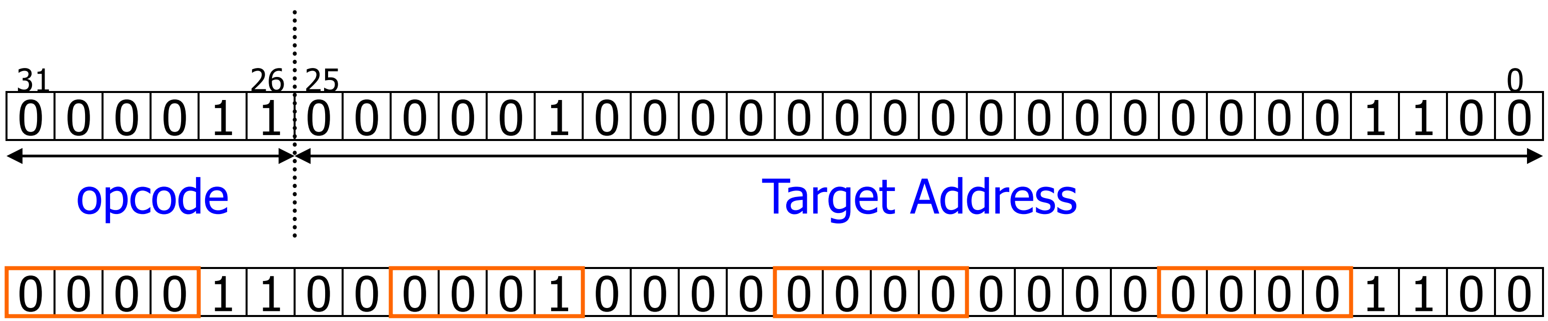
- jal will jump and push return address in $ra ($31)
- Use "jr $31" to return

Target

jal 0x00400030

$\underbrace{\text{0000 0000 0100 0000 0000 0000 0011 00}}_{\text{Target Address}}\cancel{\text{00}}$

Instruction=4 bytes



Encoding = 0x0C10000C

Control Flow

- We have: beq, bne, what about Branch-if-less-than?
- New instruction:

```
                                    if  $s1 < $s2 then
                                           $t0 = 1
    slt $t0, $s1, $s2               else
                                           $t0 = 0
```

- Can use this instruction to build "`blt $s1, $s2, Label`"
  - — can now build general control structures
- For ease of assembly programmers, the assembler allows “blt” as a “pseudo-instruction”
  - — assembler substitutes them with valid MIPS instructions
  - — there are policy of use conventions for registers

blt $4 $5 loop	⇒	slt $1 $4 $5
		bne $1 $0 loop

Constants

- Small constants are used quite frequently (50% of operands)
  e.g., A = A + 5;
  B = B + 1;
  C = C - 18;
- Solutions? Why not?
  - put 'typical constants' in memory and load them.
  - create hard-wired registers (like $zero) for constants like one.
  - Use immediate values
- MIPS Instructions:

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

How about larger constants?

- We'd like to be able to load a 32 bit constant into a register
- Must use two instructions, new "load upper immediate" instruction

```
lui $t0, 1010101010101010
```

filled with zeros

1010101010101010	0000000000000000

- Then must get the lower order bits right, i.e.,

```
ori $t0, $t0, 1010101010101010
```

	1010101010101010	0000000000000000
ori	0000000000000000	1010101010101010
	1010101010101010	1010101010101010

Input/Output

- Place proper arguments (e.g. system call code) to corresponding registers and place a 'syscall'
- Print string
  - `li   $v0, 4`
  - `la   $a0, var`
  - `syscall`
- Print integer
  - `li   $v0, 1`
  - `add $a0, $t0, $0`
  - `syscall`
- Read integer
  - `li   $v0, 5     # result in $v0`
  - `Syscall`
- See Appendix A for more.

Assembly Language vs. Machine Language

- Assembly provides convenient symbolic representation
  - much easier than writing down numbers
  - e.g., destination first
- Machine language is the underlying reality
  - e.g., destination is no longer first
- Assembly can provide 'pseudoinstructions'
  - e.g., “move $t0, $t1” exists only in Assembly
  - would be implemented using “add $t0,$t1,$zero”
- When considering performance you should count real instructions

Other Issues

- Things we are not going to cover
  support for procedures
  linkers, loaders, memory layout
  stacks, frames, recursion
  manipulating strings and pointers
  interrupts and exceptions
  system calls and conventions
- Some of these we'll talk about later
- We've focused on architectural issues
  - basics of MIPS assembly language and machine code
  - we'll build a processor to execute these instructions.

Summary of MIPS

- simple instructions all 32 bits wide
- very structured
- only three instruction formats

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

- rely on compiler to achieve performance
  — what are the compiler's goals?
- help compiler where we can

Addresses in Branches and Jumps

- Instructions:

`bne $t4,$t5,Label`	Next instruction is at Label if `$t4` ≠ `$t5`
`beq $t4,$t5,Label`	Next instruction is at Label if `$t4 = $t5`
`j Label`	Next instruction is at Label

- Formats:

I	**op**	**rs**	**rt**	**16 bit address**
J	**op**	**26 bit address**		

- Addresses are not 32 bits
  - — How do we handle this with load and store instructions?

Addresses in Branches

- Instructions:

```
bne $t4,$t5,Label
beq $t4,$t5,Label
```

bne $t4,$t5,Label — Next instruction is at Label if $\$t4 \neq \$t5$

beq $t4,$t5,Label — Next instruction is at Label if $\$t4 = \$t5$

- Formats:

	op	rs	rt	16 bit address
I				

- Could specify a register (like lw and sw) and add it to address
  - use Instruction Address Register (PC = program counter)
  - most branches are local (principle of locality)
- Jump instructions just use high order bits of PC
  - address boundaries of 256 MB

To Summarize

MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	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
Data transfer	load word	`lw $s1, 100($s2)`	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[$s2 + 100] = $s1	Word from register to memory
	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
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 less than immediate	`slti $s1, $s2, 100`	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than constant
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

Addressing Mode

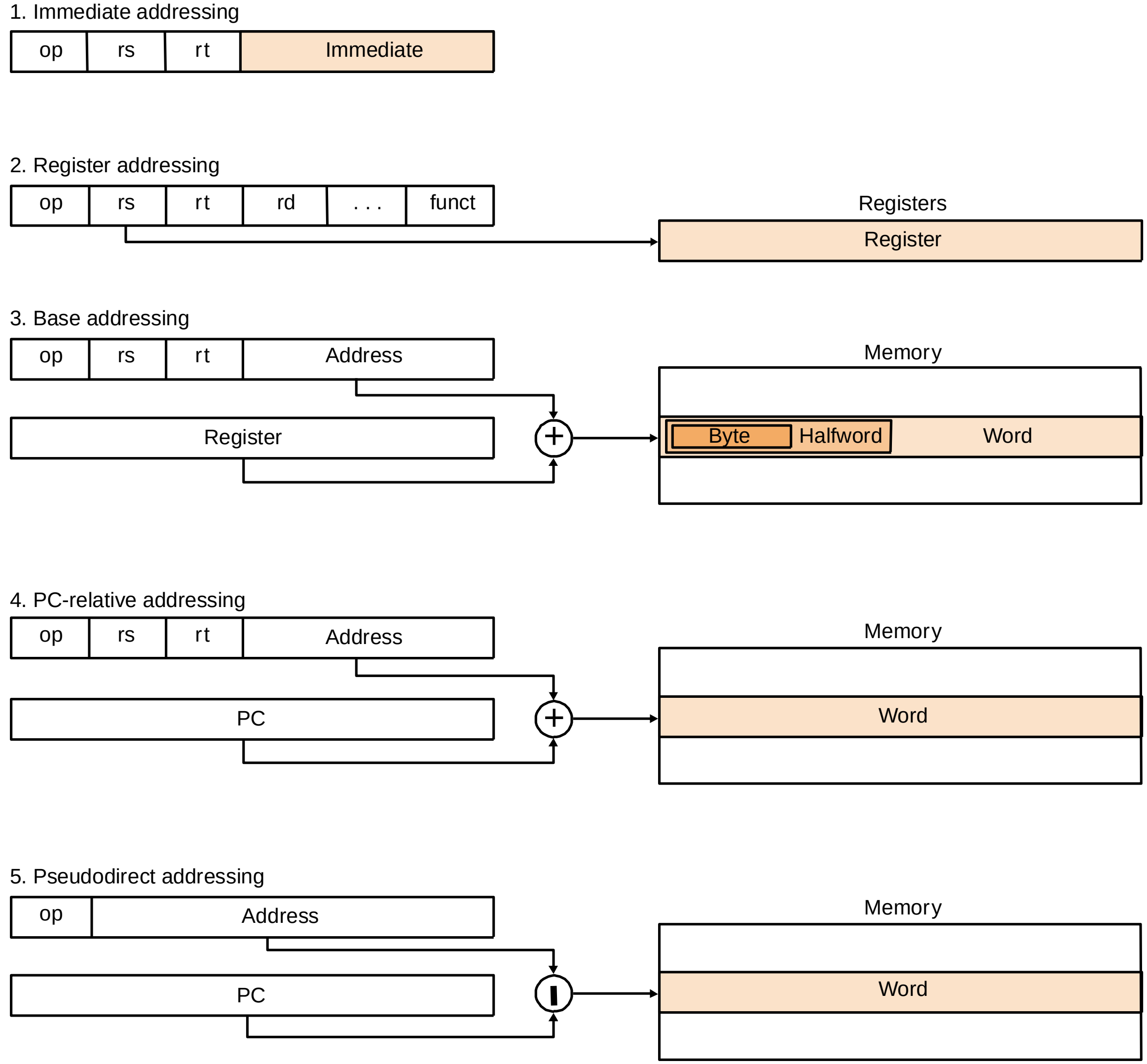
Operand is constant

Operand is in register

lb $t0, 48($s0)

bne $4, $5, Label
(label will be assembled into a distance)

j Label



Concatenation w/ PC[31..28]

Summary

- Instruction set architecture
  - a very important abstraction indeed!
- Design Principles:
  - simplicity favors regularity
  - smaller is faster
  - good design demands compromise
  - make the common case fast