Instructions: Language of the Computer

Hardware Design Principles

- *Design Principle 1:* Simplicity favours regularity
- *Design Principle 2:* Smaller is faster
- *Design Principle 3:* Good design demands good compromises
- *Design Principle 4:* Make the common case fast

Operations of the Computer Hardware

Operations of the Computer Hardware

- **MIPS** - *Microprocessor without Interlocked Pipelined Stages*)

- Computer need instructions to perform Arithmetic Operations
- MIPS architecture Provides assembly language notation
- It is a Reduced Instruction set computer (RISC) Instruction set architecture (ISA)

Operations of the Computer Hardware

Name	Example
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]

Operations of the Computer Hardware

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

Operations of the Computer Hardware

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

Operations of the Computer Hardware

- Three principles of hardware design:
  - *Design Principle 1: Simplicity favours regularity*

- 1 instruction have exactly 3 operands: no more and no less
- This keeps the hardware simple
- Hardware for a variable number of operands is more complicated than hardware for a fixed number

Operations of the Computer Hardware

- Each MIPS arithmetic instruction performs only one operation and must always have exactly three operands
- sum of two variables**(a=b+c) (need 1 instruction)**
    - **add a, b, c**
- sum of four variables **(a=b+c+d+e) (need 4 instruction)**
    - **add a, b, c** # The sum of b and c is placed in a
    - **add a, a, d** # The sum of b, c, and d is now in a
    - **add a, a, e** # The sum of b, c, d, and e is now in a

Operations of the Computer Hardware

- Higher-level programming vs Machine Level programming
- The translation from C to MIPS assembly language instructions is performed by the *compiler*

- **Compiling Two C Assignment Statements into MIPS**

HLP	MLP
a = b + c;	add a, b, c
d = a – e;	sub d, a, e

Operations of the Computer Hardware

- **Compiling a Complex C Assignment into MIPS**
    - **f = (g + h) – (i + j);**

- compiler must break this statement into several assembly instructions, since only one operation is performed per MIPS instruction

  – **add t0,g,h # temporary variable t0 contains g + h**
  – **add t1,i,j # temporary variable t1 contains i + j**
  – **sub f,t0,t1 # f gets t0 – t1, which is (g + h) – (i + j)**

Operands of the Computer Hardware

Operands of the Computer Hardware

- Operands per instruction in an Assembly language are limited
- They use special locations built directly in hardware called *registers.*
- *Registers are primitives used in hardware design*
- **MIPS architecture contain 32 registers of 32 bit long**

Operands of the Computer Hardware

- Registers are used to store operands of MIPS arithmetic instructions
- A very large number of registers may increase the clock cycle time
- Designer must balance the craving of programs for more registers with the designer’s desire to keep the clock cycle fast.
- Limited number of registers contribute to the second of our three underlying design principles of hardware technology:
    - *Design Principle 2: Smaller is faster*

Operands of the Computer Hardware

- compiler's associate program variables with registers
- EG: simple variables that contain single data elements
  - f = (g + h) – (i + j);
    - add $t0,$s1,$s2 # register $t0 contains g + h
    - add $t1,$s3,$s4 # register $t1 contains i + j
    - sub $s0,$t0,$t1 # f gets $t0 – $t1, which is (g + h)–(i + j)
  - **Complete Program**

```
Lw $s1, memg
Lw $s2, memh
Lw $s3, memi
Lw $s4, memj
add $t0,$s1,$s2
add $t1,$s3,$s4
sub $s0,$t0,$t1
Sw $s0, memf
```

Operands of the Computer Hardware

- **Memory Operands**
  - complex data structures
    - arrays and structures
    - contain many more data elements than there are registers in a computer
    - Data has to reside in register to perform operations
    - Need to transfer data between memory and register
    - **Data transfer instructions**

Operands of the Computer Hardware

- **Memory Operands**
  - Memory - large, single-dimensional array
  - Address - index to that array, starting at 0
- To access a word in memory
  - the instruction must supply the memory **address**

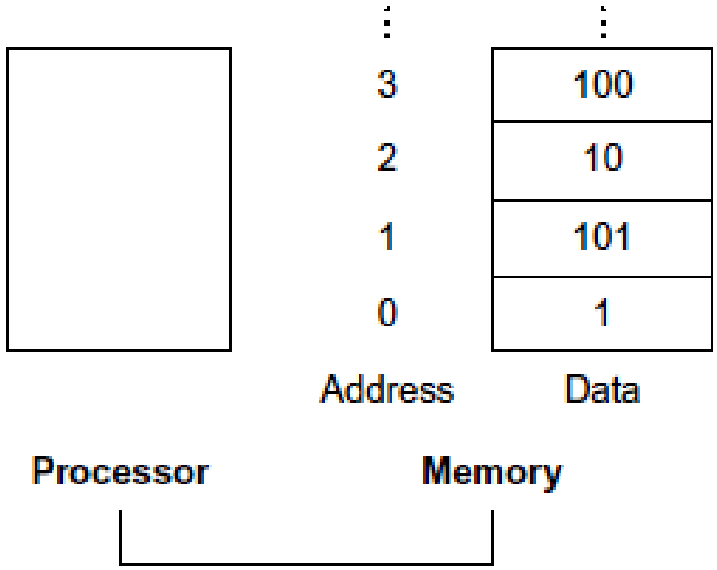


FIG Memory addresses and contents of memory at those locations

Operands of the Computer Hardware

- Data transfer instruction
  - **Load (lw – Load Word):** copies data from memory to a register
  - **Store (sw – Store word):** copies data from a register to memory
- using loads and stores compiler move variables between registers and memory
- The process of putting less commonly used variables (or those needed later) into memory is called *spilling registers*

Operands of the Computer Hardware

- Data transfer instruction
  - **EG: Operand Is in Memory**
    - g = h + A[8]
      - lw $t0,8($s3) # Temporary reg $t0 gets A[8]
      - add $s1,$s2,$t0 # g = h + A[8]
      - sw $s1, memg
  - **Note: $s3 =base address of array A**
  - **Constant 8 in lw is called offset and $s3 is base register**

Operands of the Computer Hardware

- **Memory Operands**
- Compiler while converting Program in HLL to ALL
  - Associate variables with registers
  - Associate data structures like arrays and structures to locations in memory
    - For Memory operand compiler place the proper starting address into the data transfer instructions.
    - Byte addressing is used
    - 4 bytes a word
    - Each word address differ by 4
    - words must start at addresses that are multiples of 4.
    - This requirement is called an **alignment restriction**, and many architectures have it
    - MIPS is in the big-endian camp

Operands of the Computer Hardware

- **Memory Operands**

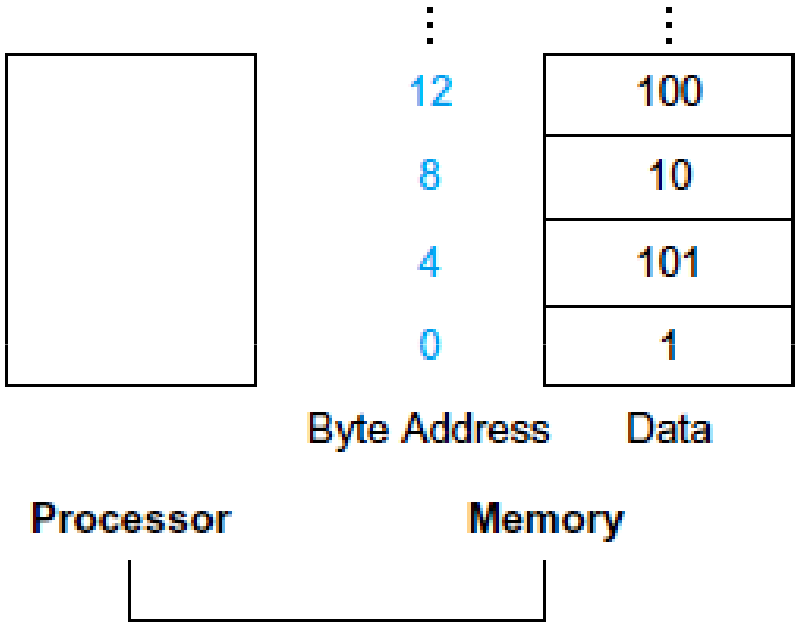


FIGURE 2.3 Actual MIPS memory addresses and contents of memory for those words

Operands of the Computer Hardware

- **Memory Operands**
  - **EG: Compiling Using Load and Store**
    - A[12] = h + A[8];

    - lw $t0,32($s3) # Temporary reg $t0 gets A[8]
    - add $t0,$s2,$t0 # Temporary reg $t0 gets h + A[8]
    - sw $t0,48($s3) # Stores h + A[8] back into A[12]
- **Note: A[8] $s3 contain base address of A**
- **A[8] is the 9th word starts at 4*8 =32**
- **A[12] 12*8= 48**

Address	**0**	**4**	**8**	**12**	**16**	**20**	**24**	**28**	**32**
Dara	0	1	2	3	4	5	6	7	8

Operands of the Computer Hardware

- **Memory Operands**
- Relating size and speed
- Memory must be slower than registers
- Memory must be larger than registers
- A MIPS arithmetic instruction can read two registers, operate on them, and write the result.
- A MIPS data transfer instruction only reads one operand or writes one operand, without operating on it.

Operands of the Computer Hardware

- **Memory Operands**
- Data is more useful when in a register
- Registers take less time to access *and have higher throughput than memory*
- Data in registers are faster to access and simpler to use
- Accessing registers also uses less energy than accessing memory
- To achieve highest performance and conserve energy, an instruction set architecture must have a sufficient number of registers, and compilers must use registers efficiently.

Operands of the Computer Hardware

- **Constant or Immediate Operands**
- Program will use a constant in an operation
  - EG: incrementing an index to point to the next element of an array

  - EG 1: to add the constant 4 to register $s3 $s3= $s3+4
    - lw $t0, AddrConstant4($s1) # $t0 = constant 4
    - add $s3,$s3,$t0 # $s3 = $s3 + $t0 ($t0 == 4)
    - Note : Location of value 4 is $s3+ AddrConstant4

  - EG 2: : to add the constant 4 to register $s3 $s3= $s3+4
    - addi $s3,$s3,4 # $s3 = $s3 + 4
    - Note addi is a versions of the arithmetic instructions in which one operand is a constant

Operands of the Computer Hardware

- **Constant or Immediate Operands**
- Constant operands
  - used frequently
  - including constants inside arithmetic instructions, operations are much faster and use less energy than if constants were loaded from memory.

Operands of the Computer Hardware

- **Constant or Immediate Operands**
- The constant zero
  - simplify the instruction set by offering useful variations
  - EG: the move operation is just an add instruction where one operand is zero
  - Hence, MIPS dedicates a register $zero to be hard-wired to the value zero.
  - Using frequency to justify the inclusions of constants is another example of the great idea of making the **common case fast**
  - **(avoid optimizing hard part try to make the frequent things faster)**

Immediate Operands

- Constant data specified in an instruction

  ```
  addi $s3, $s3, 4
  ```

- No subtract immediate instruction
  - Just use a negative constant

    ```
    addi $s2, $s1, -1
    ```

- *Design Principle 4:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

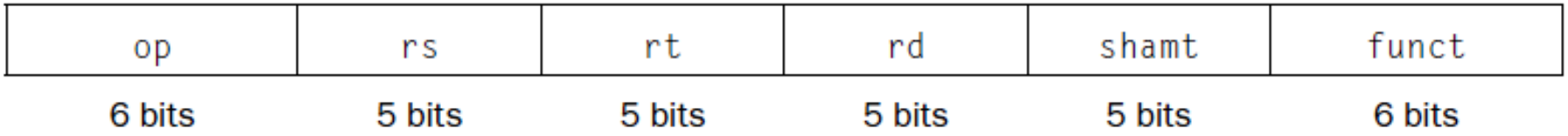
Representing Instructions in the Computer

Representing Instructions

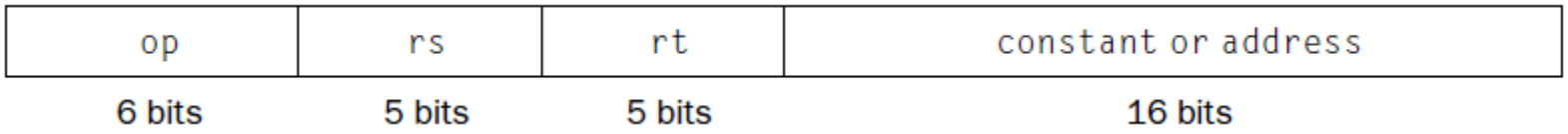
- Instructions
  - Stored in memory
  - a series of high and low electronic signals
  - represented as numbers
  - Instructions, registers are mapped to numbers
    - Registers $s0 to $s7 map to 16 to 23
    - Registers $t0 to $t7 map to 8 to 15

Instruction Format Types

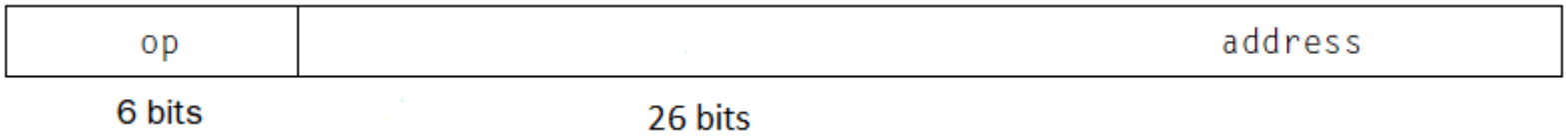
- R-Type



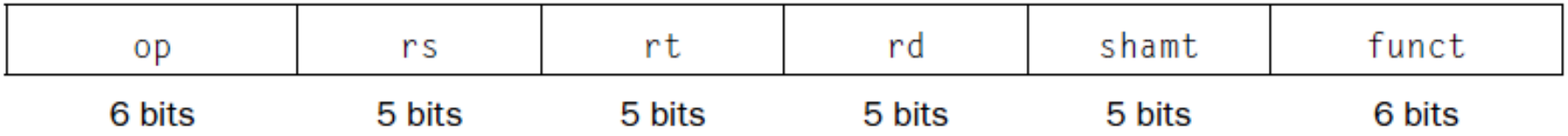
- I-Type



- J-Type



Representing Instructions R-Type



- *op:* Basic operation of the instruction, traditionally called the opcode.
- *rs:* The first register source operand.
- *rt:* The second register source operand.
- *rd:* The register destination operand. It gets the result of the operation.
- *shamt:* Shift amount.

- *funct:* Function. This field, often called the *function code*, selects the specific variant of the operation in the op field.

R-format Example

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

$00000010001100100100000000100000_2 = 02324020_{16}$

Representing Instructions

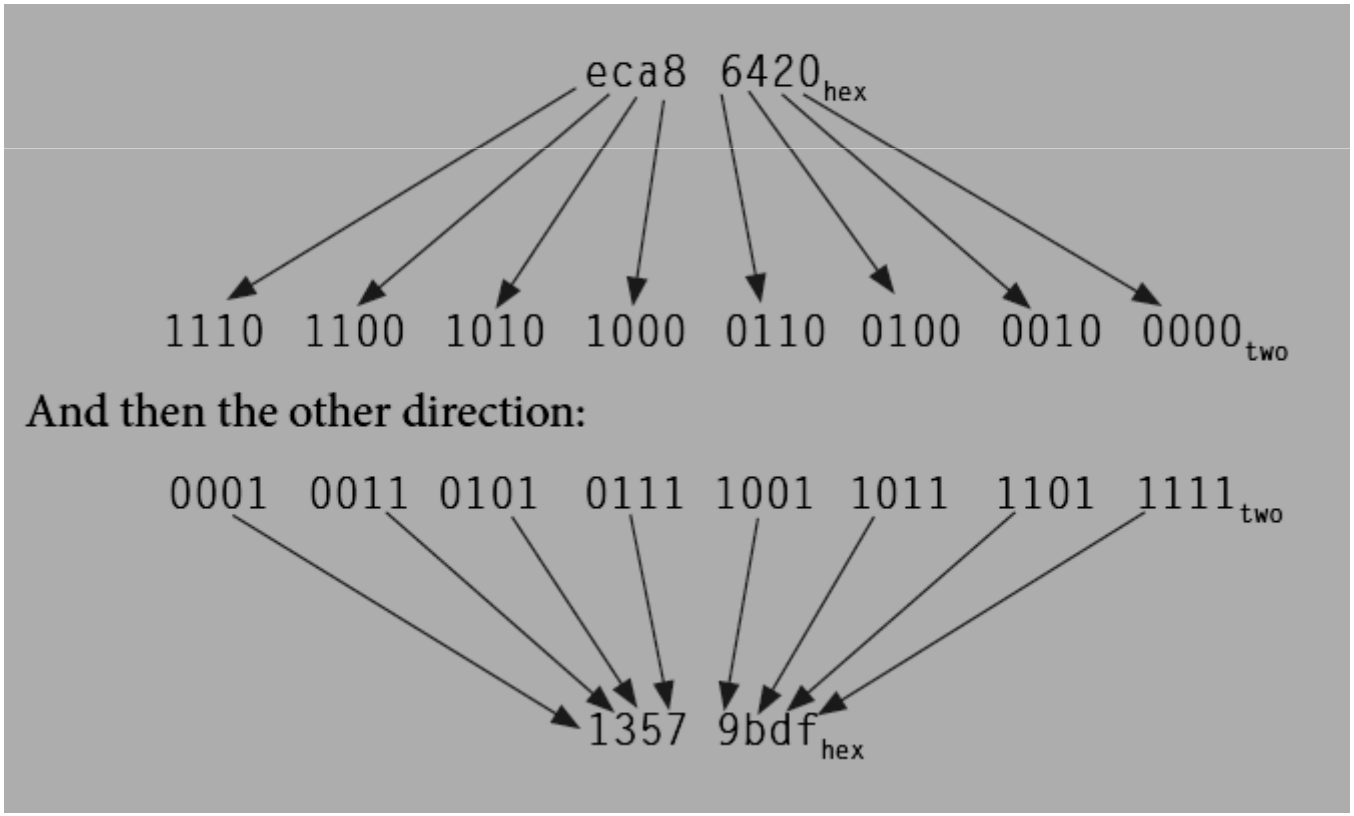
- Assembly lang. to Machine lang.
- Mnemonic to numeric
- Instruction Format – 32 bits (word size)
  - **Eg:** add $t0,$s1,$s2

Instruction	add	$s1	$s2	$t0		
Decimal	0	17	18	8	0	32
Binary	000000	10001	10010	01000	00000	100000
No. of bits	6 bits	5 bits	5 bits	5 bits	5 bits	6bits

  - **Fields:**
    - Each segments of an instruction is called a *field*
    - *The first and* last fields 0 and 32 tell addition
    - The second & third fields - two source operand
    - The fourth fields - destination operand
    - The fifth field – unused – so set to 0

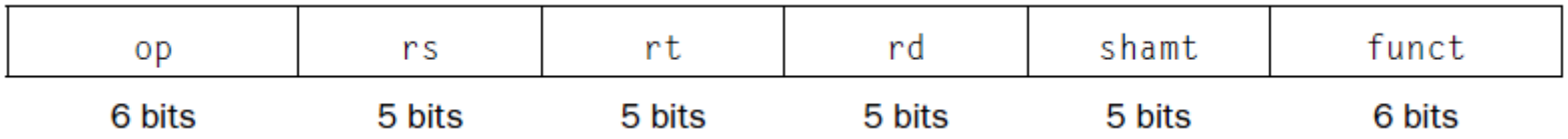
Representing Instructions

- Numeric version of instructions **machine language**
- A sequence of such instructions ***machine code***
- Machine instructions are represented using **hexadecimal code**

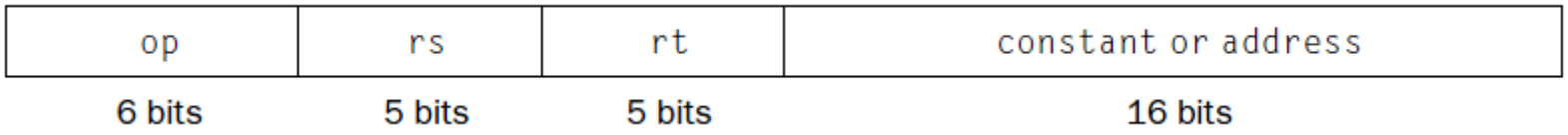


Representing Instructions

- Conflict
- This conflict leads to the final hardware design principle
- *Design Principle 3: Good design demands good compromises*
- The compromise chosen by the MIPS designers is to keep all instructions the same length
- So there is a need for different kinds of instruction formats for different kinds of instructions
- IF
  - *R-type (for register)* or *R-format.*



  - *I-type (for immediate)* or *I-format*



Representing Instructions

- Conflict
  - Between the desire to keep all instructions the same length and the desire to have a single instruction format
- Eg: lw $t0, 8($s1)

- Need to specify two registers and a constant
- If one 5 bit filed is used for constant then the constant may vary from 0 to 2^5 : only 32 at the maximum
- This 5-bit field is too small to be useful
- So the desire to have a single instruction format will not be helpful