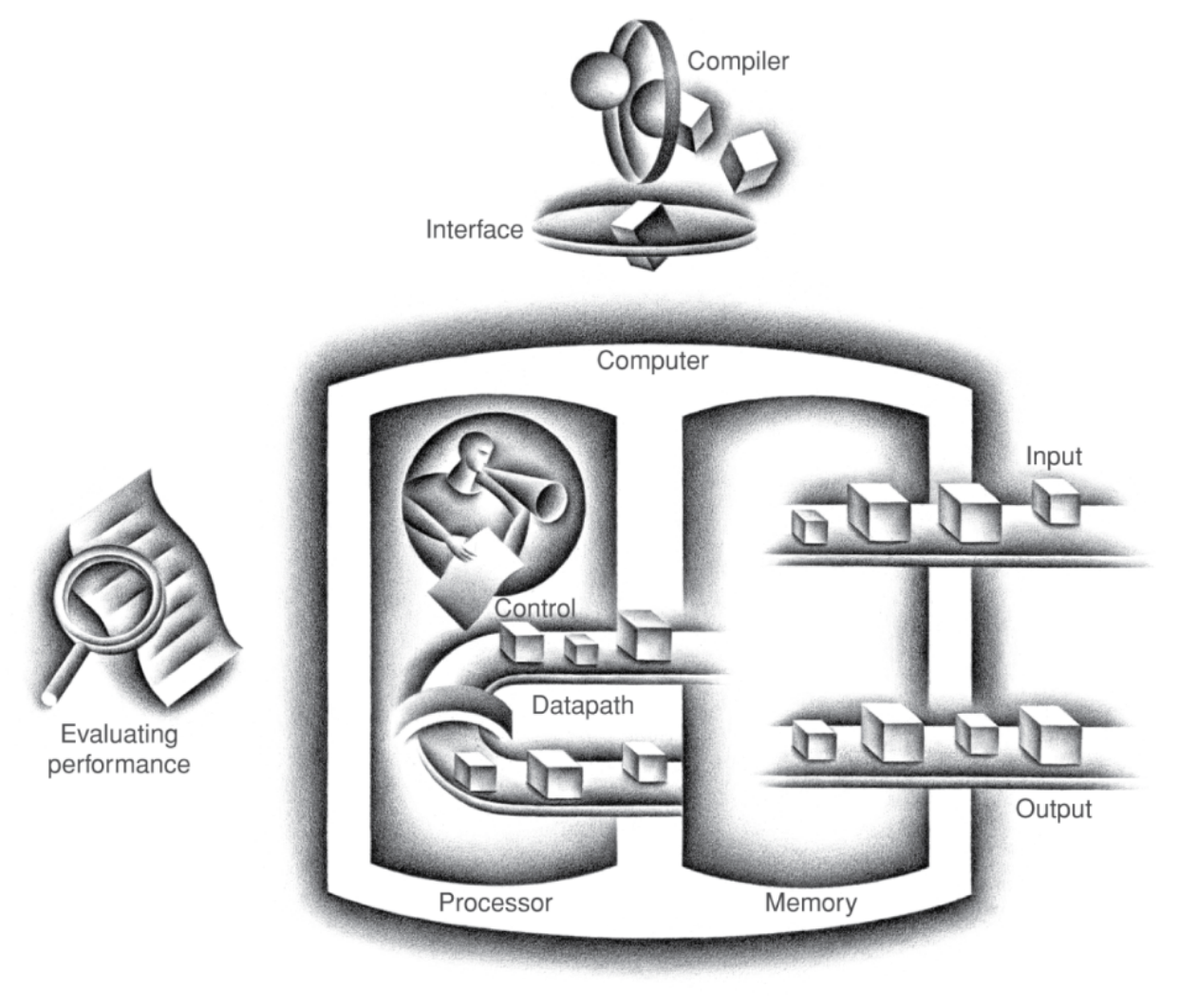
II. Instructions Sets

Computer Architecture

TDTS10

Ahmed Rezine
(Based on slides of Unmesh D. Bordoloi)

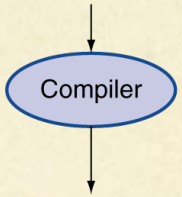
Components of a Computer



Hierarchical Layers of Program Code

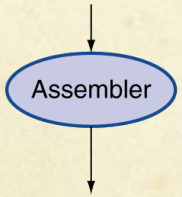
High-level language program (in C)

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



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
```



Binary machine language program (for MIPS)

```
00000000101000010000000000011000
00000000000011000000011000001000 01
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

Instruction Set

- The repertoire of instructions of a computer
- Computers can implement the same instruction set in different manners
- Instruction sets can be quite different, but they have many aspects in common

The MIPS Instruction Set

- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com) in the 80s
- Large share of embedded core market
  - Applications in consumer electronics, network/ storage equipment, cameras, printers, ...
- Typical of many modern ISAs

Instruction Set

- Stored-program concept
  - The idea that instructions and data of many types can be stored in memory as numbers, leading to stored-program computer

- Let us look into MIPS instruction set one by one to understand this

Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination

```
add a, b, c

# a gets b + c
```

Arithmetic Operations

- Operand is a quantity on which an operation is performed

```
add a, b, c
```

- How many operands in this instruction?

- All arithmetic operations have this form

Design Principle 1

- All arithmetic operations have same form
- *Design Principle 1:* Simplicity favors regularity
  - Regularity makes hardware implementation simpler

Arithmetic Example

- C code:

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

- Compiled MIPS code: ?

- Hints:
  - Use sub instructions, e.g., a=b-c, is sub a,b,c
  - Use two temporary variables t0 and t1

Arithmetic Example

- C code:

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

- Compiled MIPS code:

```
add t0, g, h   # temp t0 = g + h
add t1, i, j   # temp t1 = i + j
sub f, t0, t1  # f = t0 - t1
```

Register Operands

- The operands of arithmetic instructions must be from special location in hardware called *registers*

- *Registers* are primitives of hardware design and are visible to programmers

Register Operands

- Assembler names
  - **$t0, $t1, ..., $t9** for temporary values
  - **$s0, $s1, ..., $s7** for saved variables

Register Operand Example

- Compiler's job to associate variables of a high-level program with registers

- C code:

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

  - f, ..., j in $s0, ..., $s4

- Compiled MIPS code ?

Register Operand Example

- Compiled MIPS code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

Register Operands

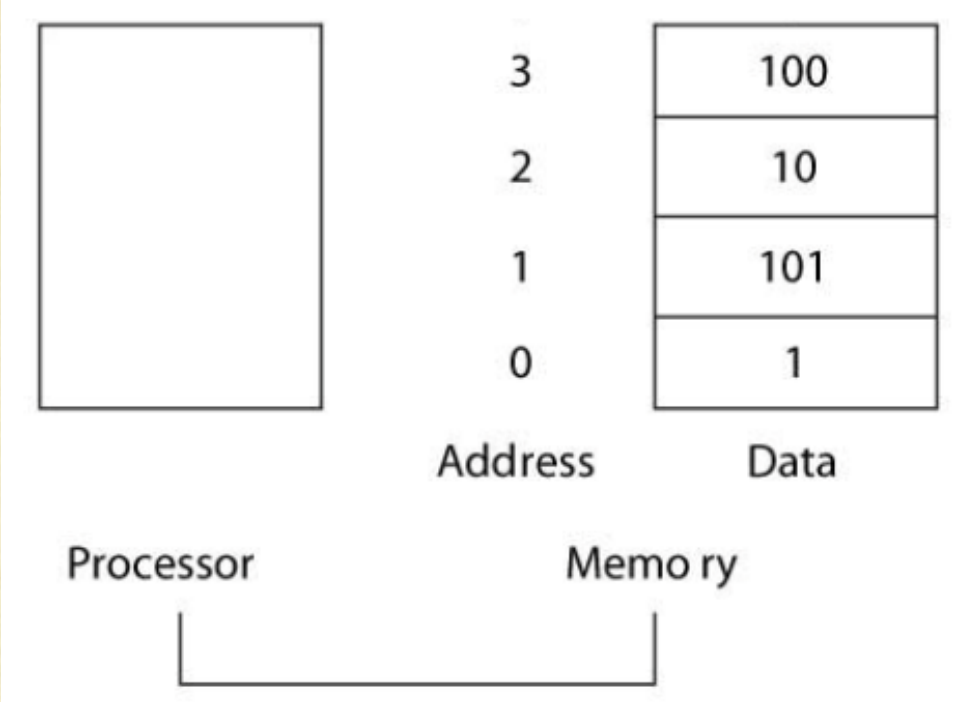
- MIPS has a 32 × 32-bit register file
  - Numbered 0 to 31
  - 32-bit data called a "word"
- Word is the natural unit of access, typically 32 bits, corresponds to the size of a register in MIPS
- There may be only 3 operands and they must be chosen from one of the 32 registers. Why only 32 ?

Design Principle 2

- Smaller is faster
  - Larger registers will increase clock cycle time -- electronic signals take longer when they travel farther
- Design principles are not hard truths but general guidelines
  - 31 registers instead of 32 need not make MIPS faster

Memory Operands

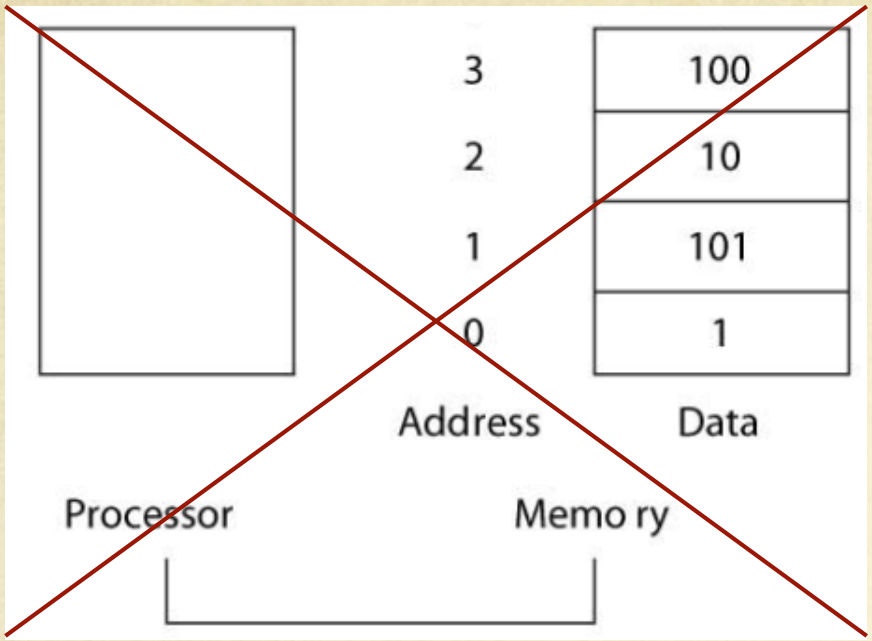
- Programming languages, C, Java, ...
  - Allow complex data structures like arrays and lists
  - More data elements than the number of registers in a computer
  - Where are they stored ?
- But, arithmetic operations are applied on register operands
- Hence, data transfer instructions are required to transfer data from memory to registers
  - Load values from memory into registers
  - Store result from register to memory

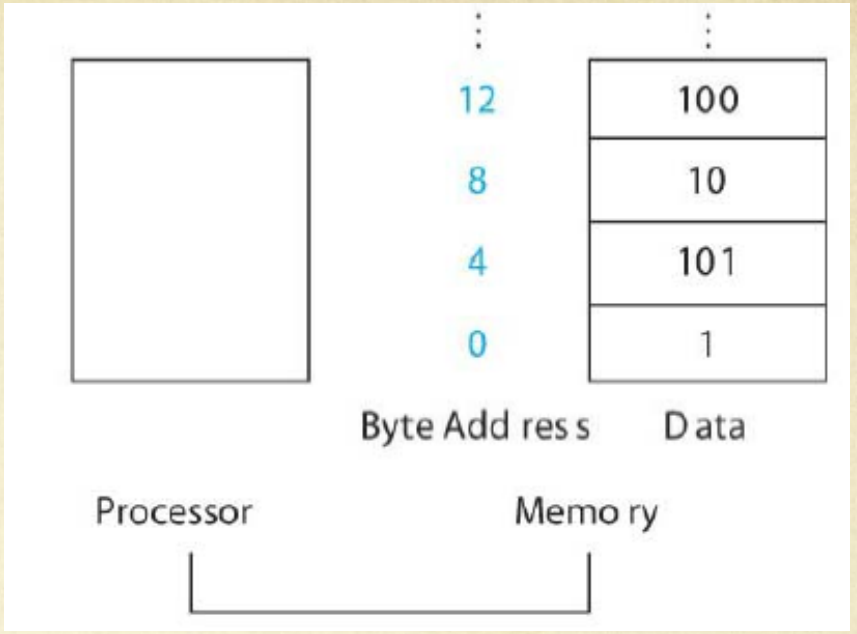


- Memory is like an array
- Data transfer instructions must supply the address (index/offset) of the memory (array)

Memory Operands

- Memory is byte addressed
  - Each address identifies an 8-bits byte
- Words are aligned in memory
  - Each word is 32 bits or 4 bytes
  - To locate words, addresses are in multiples of 4





- A is an array of words
- What is the offset to locate A[8] ?
  - A[0] – 0
  - A[1] – 4
  - A[2] – 8
  - …
  - A[8] – 32

Memory Operands

- Why is memory not word-addressable?

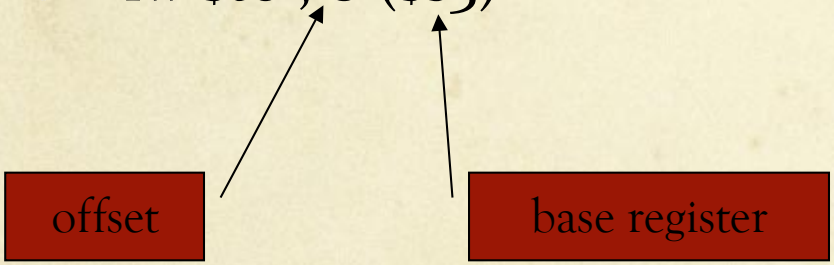
Memory Operands

Why is memory not word-addressable?

- Bytes are useful in many programs.
- In a word addressable system, it is necessary to:
  - compute the address of the word containing the byte,
  - fetch that word,
  - extract the byte from the two-byte word.
- Although the processes for byte extraction are well understood, they are less efficient than directly accessing the byte.
- For this reason, many modern machines are byte addressable.

Memory Operands - lw

- *Load* instruction
  - lw refers to *load word*
  - lw registerName, offset (registerWithBaseAddress)
  - lw $to , 8 ($s3)



Memory Operand Example 1

- C code:

```
g = h + A[8];
```

  - g in **$s1**
  - h in **$s2**
  - base address of A in **$s3**
  - A is an array of 100 words

- Compiled MIPS code ?

Memory Operand Example 1

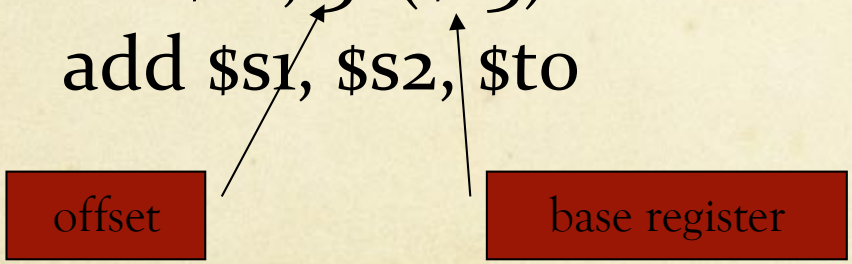
- C code:

```
g = h + A[8];
```

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

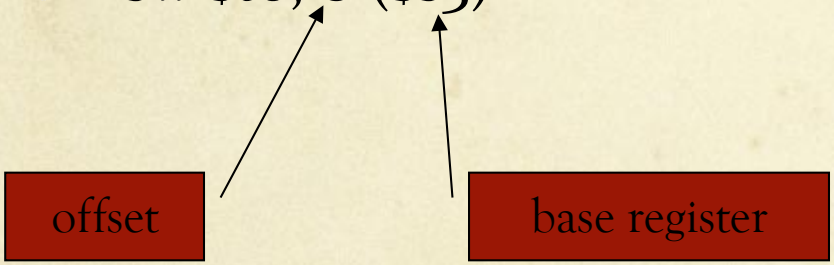
- Compiled MIPS code:

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



Memory Operands - sw

- *Store* instruction
  - sw refers to *store word*
  - sw registerName, offset (registerWithBaseAddress)
  - **sw $t0, 8 ($s3)**



Memory Operand Example 2

- C code:

```
A[12] = h + A[8];
```

  - h in **$s2**, base address of A in **$s3**

- Compiled MIPS code:

Memory Operand Example 2

- C code:

```
A[12] = h + A[8];
```

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

- Compiled MIPS code:
  - Index 8 requires offset of 32

```
lw  $t0, 32($s3)      # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)      # store word
```

Registers vs. Memory

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

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 Principle3

- Make the common case fast
  - Small constants are common : immediate operand avoids a load instruction
  - Allows us to avoid using memory meaning faster operations and lesser energy

The Constant Zero

- MIPS register 0 (**$zero**) always holds the constant 0
  - Cannot be overwritten
- Useful for common operations
  - E.g., move between registers

    add $t2, $s1, $zero

Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- MIPS instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, ...
  - Regularity!
- Register numbers
  - **$to – $t7** are reg's 8 – 15
  - **$so – $s7** are reg's 16 – 23

Example

```
add $t0, $s1, $s2
```

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

$00000010001100100100000000100000_2$

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

Representing Instructions

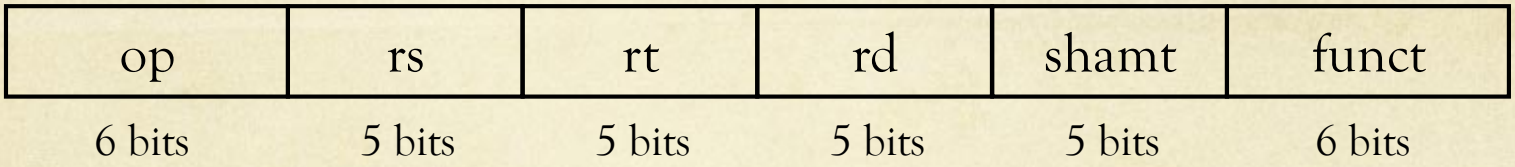
- The layout or the form of representation of instruction is composed of fields of binary numbers

- The numeric version of instructions is called machine language and a sequence of such instructions is called machine code

Instruction types

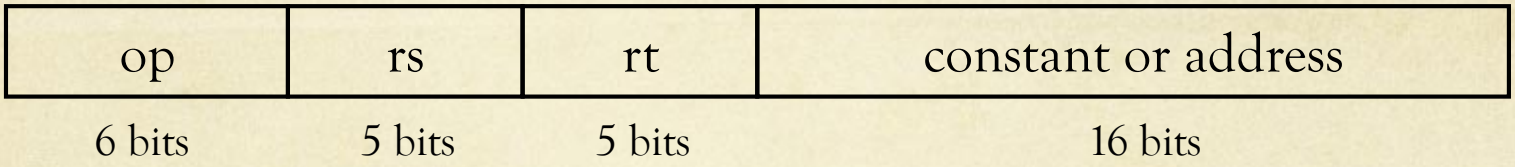
- R format (for register)
  - add, sub
- I-format (for immediate)
  - Immediate
  - Data transfer

MIPS R-format Instructions



- Instruction fields
  - 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)

MIPS I-format Instructions



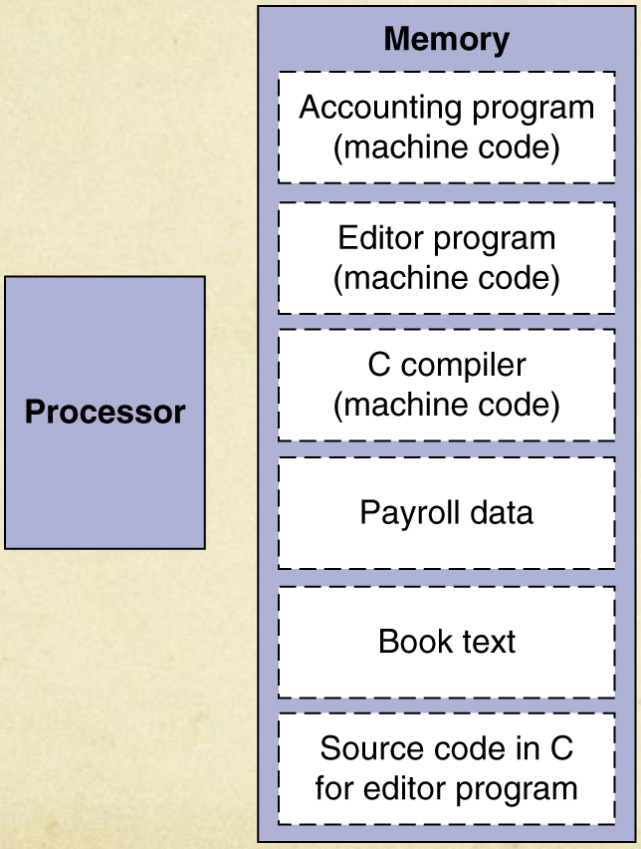
- Immediate arithmetic and load/store instructions
  - rt: destination register number
  - rs: register number with address
  - Constant/ Address: offset added to base address in rs

Design Principle 4

- Ideally,
  - Keep all instructions of the same format and length
  - But this makes it difficult to address large memories
    - Compromise and allow different formats

- Principle 4: Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible
    - See example in page 98

Stored Program Computers



- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, ...
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

Logical Operations

- 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

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

Shift Operations

- Shift left logical
  - Shift bits to the left and fill the empty bits with zeros
  - **sll $t2,$s0,3**

0000 0000 0000 0000 0000 0000 0000 0001

Shift Operations

- Shift left logical
  - Shift bits to the left and fill the empty bits with zeros
  - sll $t2,$s0,3

```
0000 0000 0000 0000 0000 0000 0000 0001
```

```
0000 0000 0000 0000 0000 0000 0000 1000
```

Shift Operations

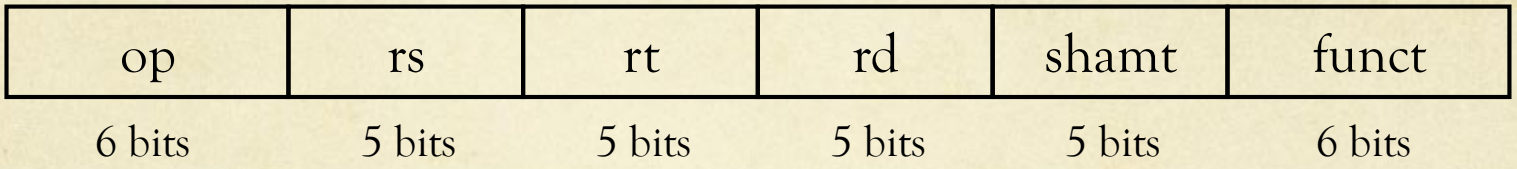
- Shift left logical
  - Shift bits to the left and fill the empty bits with zeros
  - sll $t2,$s0,3

```
0000 0000 0000 0000 0000 0000 0000 0001
```

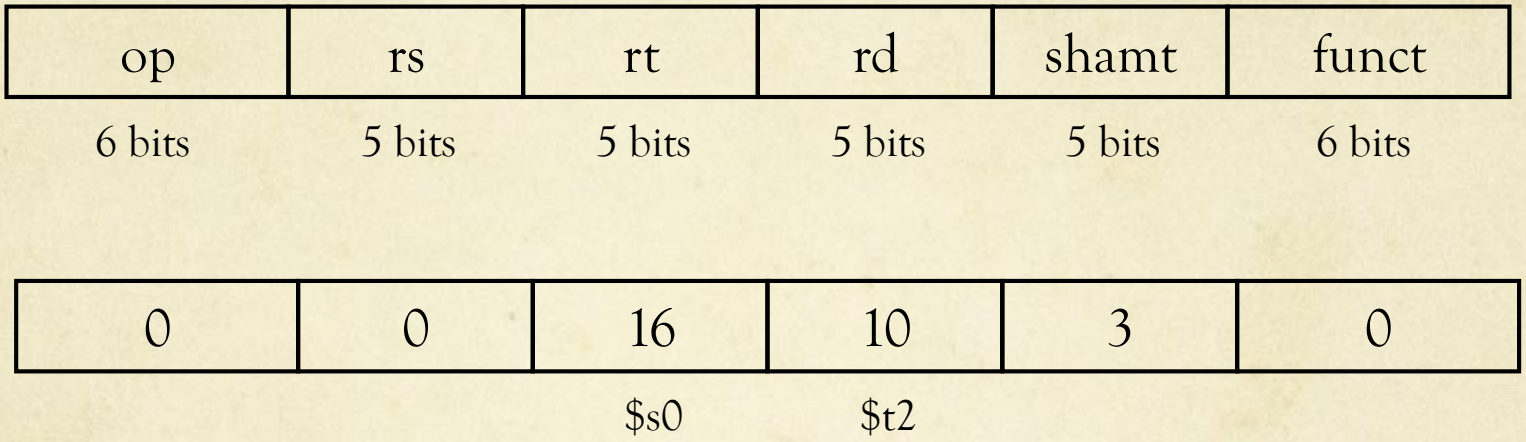
```
0000 0000 0000 0000 0000 0000 0000 1000
```

  - `sll` by *i* bits multiplies by 2^i

Shift Operations



Shift Operations

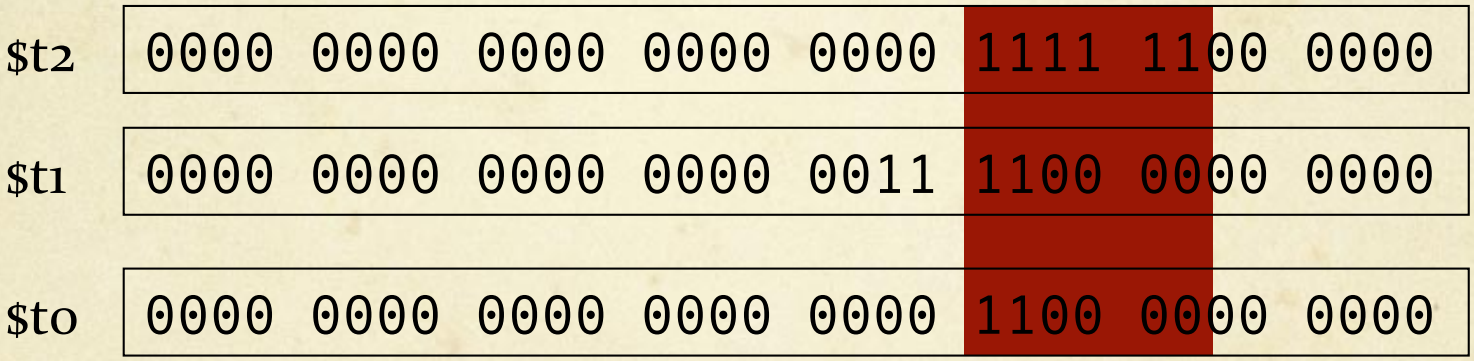


- **sll $t2,$s0,3**
- shamt: how many positions to shift
- Similarly, …
  - Shift right logical

AND Operations

- Mask bits in a word
  - Select some bits, clear others to 0

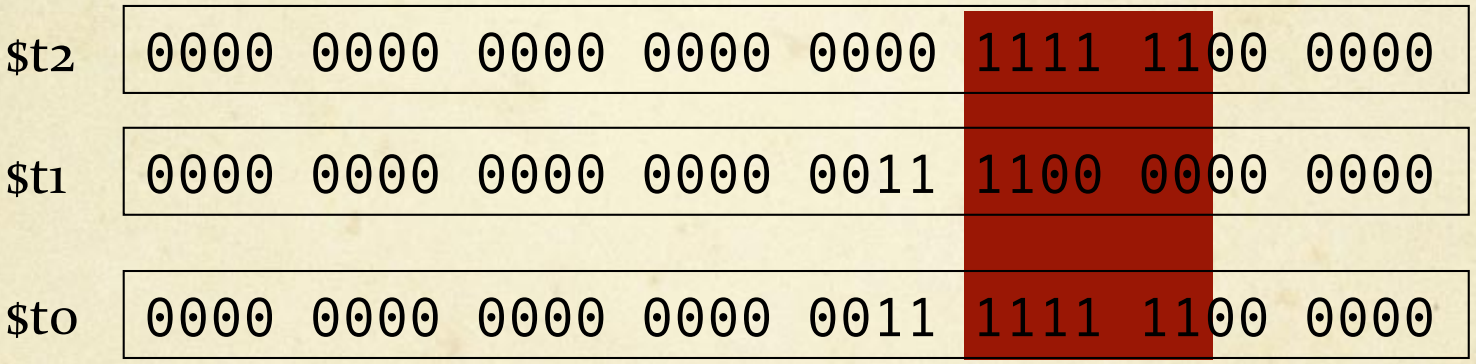
and $t0, $t1, $t2



OR Operations

- Include bits in a word
  - Set some bits to 1, leave others unchanged

or $t0, $t1, $t2



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 $to, $t1, $zero ← Register 0: always read as zero

$t1	0000 0000 0000 0000 0011 1100 0000 0000
$to	1111 1111 1111 1111 1100 0011 1111 1111

Shift Operations

- Shift right arithmetic
  - Shift bits to the right and fill the empty bits with sign bit
  - **sra $t2,$s0,shamt**

- Shift left arithmetic can be achieved with sll!

Conditional Operations

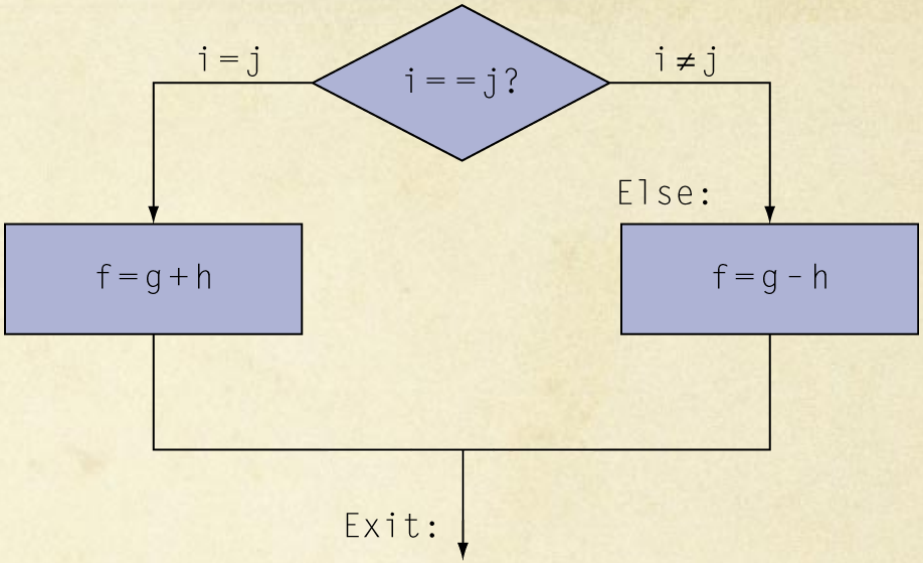
- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs, rt, L1`
  - if (rs == rt) branch to instruction labeled L1;
- `bne rs, rt, L1`
  - if (rs != rt) branch to instruction labeled L1;
- `j L1`
  - unconditional jump to instruction labeled L1

Compiling If Statements

- C code:

```
if (i==j) f = g+h;
else f = g-h;
```

  - f, g, … in $s0, $s1, …

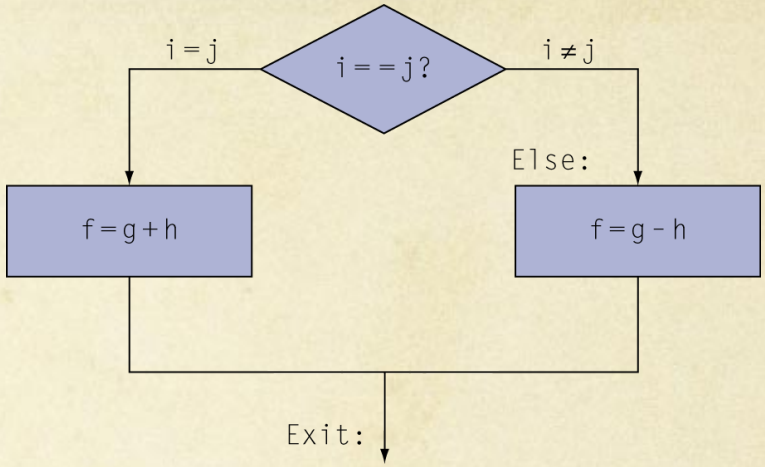


Compiling If Statements

- C code:

```
if (i==j) f = g+h;
else f = g-h;
```

  - f, g, ... in $s0, $s1, ...



- Compiled MIPS code:

```
      bne $s3, $s4, Else
      add $s0, $s1, $s2
      j   Exit
Else: sub $s0, $s1, $s2
Exit: ...
```

Assembler calculates addresses

RISC vs CISC

CISC Approach

- Complex Instruction Set Computer
- C code:

```
g = h + A[8];
```

- CISC

  add a,32<b>

- Achieved by building complex hardware that loads value from memory into a register and then adds it to register a and stores the results in register a

CISC vs RISC

- C code:

```
g = h + A[8];
```

- CISC

  add a,32<b>

- Compiled MIPS code:

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

CISC Advantages

- Compiler has to do little
  - Programming was done in assembly language
  - To make it easy, more and more complex instructions were added
- Length of the code is short and hence, little memory is required to store the code
  - Memory was a very costly real-estate
- E.g., Intel x86 machines powering several million desktops

RISC Advantages

- Each instruction needs only one clock cycle
- Hardware is less complex

RISC Roadblocks

- RISC processors, despite their advantages, took several years to gain market
  - Intel had a head start of 2 years before its RISC competitors
  - Customers were unwilling to take risk with new technology and change software products
  - They have a large market and hence, they can afford resources to overcome complexity
- Interview with Hennessy
  - http://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/about/interview.html

Concluding Remarks

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises
- Layers of software/hardware
  - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs
  - c.f. x86

Supporting procedures

- Procedure or a function is a tool used to structure programs
  - It may have arguments
  - It may return a value
- E.g.,
  - ```
    void swap(int v[], int k)
    {
      int temp;
      temp = v[k];
      v[k] = v[k+1];
      v[k+1] = temp;
    }
    ```
    ```

Execution of a procedure

- Steps required
  1. Place parameters (arguments) in registers
  2. Transfer control to procedure
  3. Acquire storage resources for procedure
  4. Perform procedure's operations
  5. Place result in register for caller
  6. Return to place of call

Register Usage

- $a0 – $a3: save the arguments (reg's 4 – 7)
- $v0, $v1: save the result values to be returned (reg's 2 and 3)
- $ra: return address (reg 31)

Procedure Call Instructions

- Procedure call: jump and link

```
jal ProcedureLabel
```

  - Address of following instruction (return address) put in $ra
  - Jumps to target address

- Procedure return: jump register

```
jr $ra
```

  - Copies $ra to program counter or PC (What is PC?)
  - Can also be used for computed jumps
    - e.g., for case/switch statements

Stack

- What if there are more than four arguments?
- Spill registers to stack
- Stack is a data structure – a Last-in-First-out queue that is ideal for this
- More on the *MIPS calling convention* in your lession – part of exam syllabus !