Lecture 3: Instructions - Language of the Computers

CSCE2610 Computer Organization

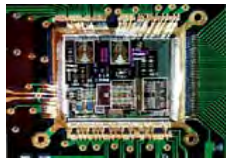
Instructor: Saraju P. Mohanty, Ph. D.

NOTE: The figures, text etc included in slides are borrowed from various books, websites, authors pages, and other sources for academic purpose only. The instructor does not claim any originality.

What are Instructions?

- 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
  - similar to other architectures developed since the 1980's
  - used by NEC, Nintendo, Silicon Graphics, Sony

- *Design goals:*
  - *maximize performance*
  - *minimize cost*
  - *reduce design time*



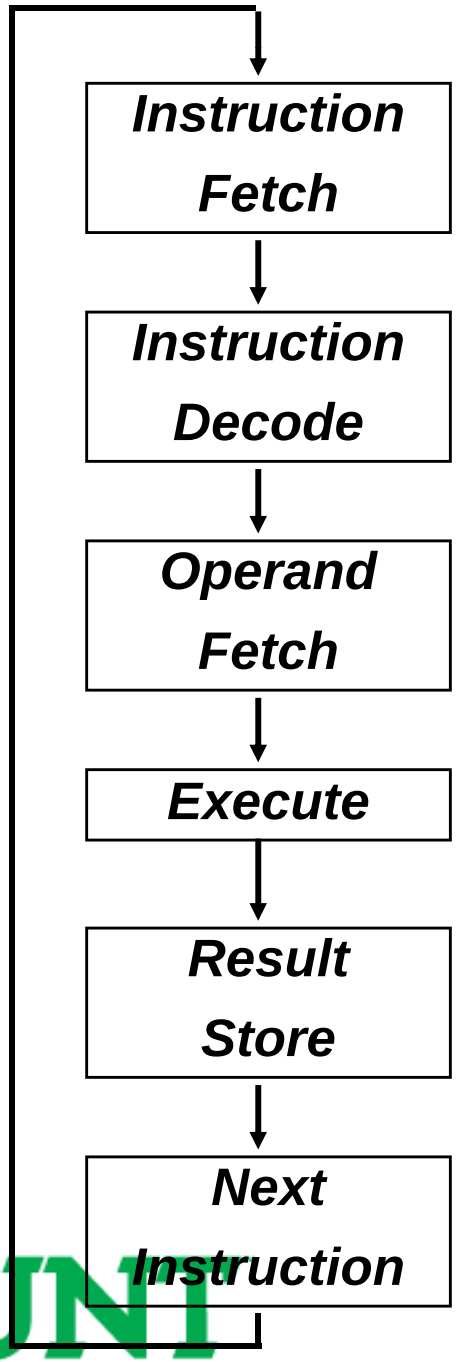
Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

The MIPS Instruction Set

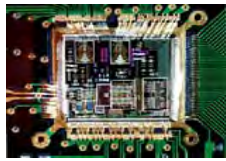
- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern ISAs
  - See MIPS Reference Data tear-out card, and Appendixes B and E

Instruction Set Architecture: What Must be Specified?



- Instruction Format or Encoding
  - how is it decoded?
- Location of operands and result
  - where other than memory?
  - how many explicit operands?
  - how are memory operands located?
  - which can or cannot be in memory?
- Data type and Size
- Operations
  - what are supported
- Successor instruction
  - jumps, conditions, branches

- fetch-decode-execute is implicit!



Instruction Categories in MIPS Processor

- Arithmetic
- Logical
- Data Transfer
- Conditional Branch
- Unconditional Branch

Design Principles

- Instruction complexity is only one variable
  - lower instruction count vs. higher CPI (cycles per instruction) / lower clock rate.
- Design Principles:
  - simplicity favors regularity
  - smaller is faster
  - make the common case fast
  - good design demands compromise
- Instruction set architecture
  - a very important abstraction indeed!

Arithmetic Operations

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

```
add a, b, c   # a gets b + c
```

- All arithmetic operations have this form
- *Design Principle 1:* Simplicity favours regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Arithmetic Example

- C code:

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

- Compiled MIPS code:

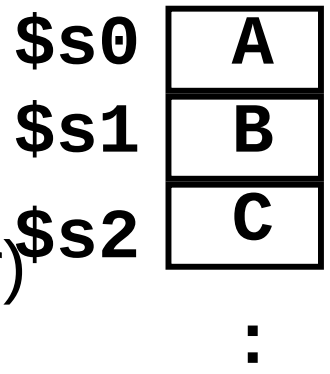
```
add t0, g, h   # temp t0 = g + h
add t1, i, j   # temp t1 = i + j
sub f, t0, t1  # f = t0 - t1
```


MIPS Arithmetic

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

C code: A= B + C

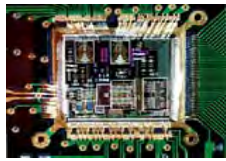
MIPS code: add $s0, $s1, $s2
(associated with variables by compiler)



Note:

(1) "$s0" represents a register

(2) Variables A, B, C are stored in registers $s0, $s1, and $s2, respectively.



MIPS Arithmetic

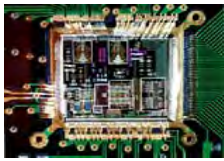
- Design Principle: simplicity favors regularity. Why?
- Of course this complicates some things...

C code: A = B + C + D;
E = F - A;

MIPS code: add $t0, $s1, $s2
add $s0, $t0, $s3
sub $s4, $s5, $s0

Note: register $t0, $t1 are temporary registers

- Operands must be registers, only 32 registers provided
- *Design Principle 2*: smaller is faster. Why?



Register Operands

- Arithmetic instructions use register operands
- MIPS has a 32 × 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a “word”
- Assembler names
  - $t0, $t1, …, $t9 for temporary values
  - $s0, $s1, …, $s7 for saved variables
- *Design Principle 2:* Smaller is faster
  - c.f. main memory: millions of locations

Register Operand Example

- C code:

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

  - f, …, j in $s0, …, $s4

- Compiled MIPS code:

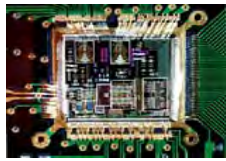
```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```


Logical Operations

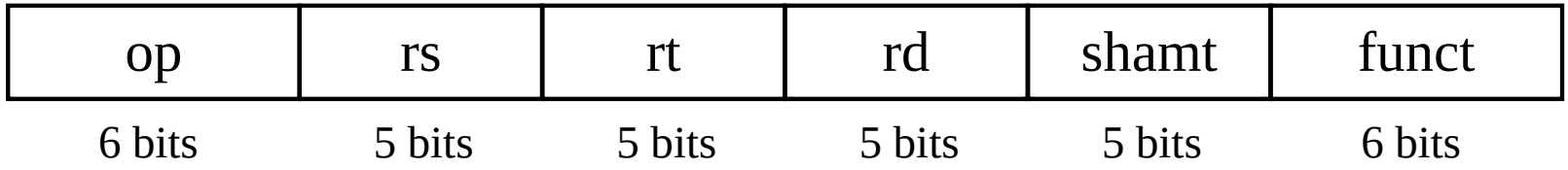
- Instructions for bitwise manipulation

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

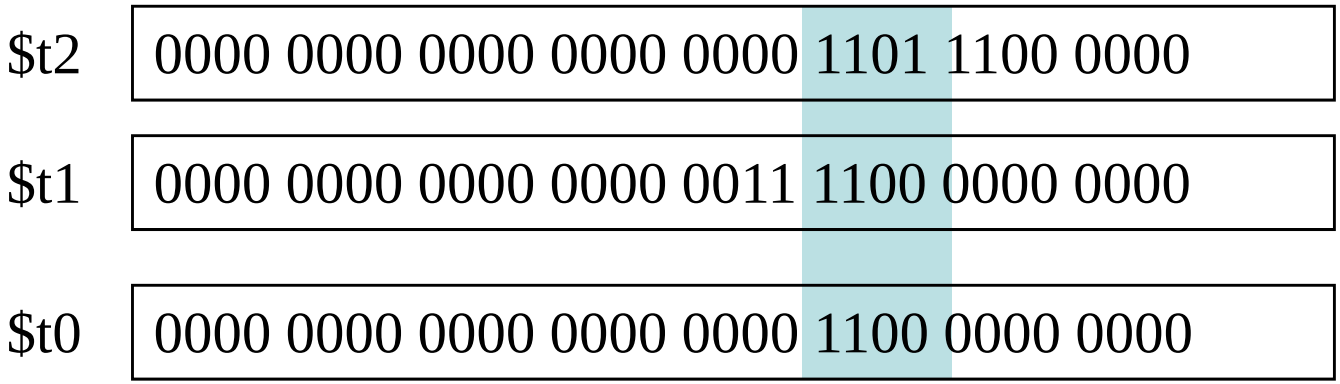


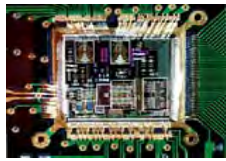
- shamt: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - `sll` by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - `srl` by *i* bits divides by 2^i (unsigned only)

AND Operations

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

```
and $t0, $t1, $t2
```

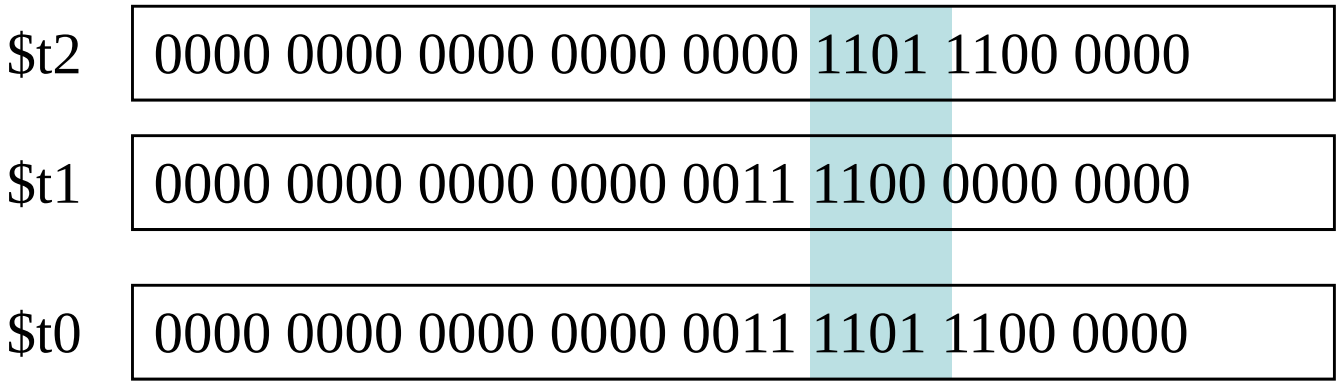




OR Operations

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

```
or $t0, $t1, $t2
```



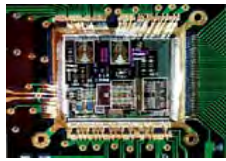
NOT Operations

- 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 $t0, $t1, $zero
```

Register 0: always read as zero

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



Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4
- MIPS is Big Endian
  - Most-significant byte at least address of a word
  - *c.f.* Little Endian: least-significant byte at least address

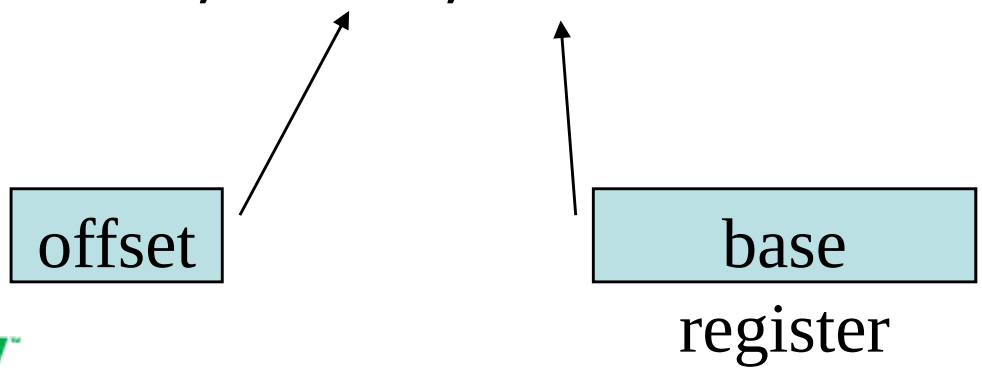
Memory Operand Example 1

- C code:

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

  - g in $s1, h in $s2, base address of A in $s3
- Compiled MIPS code:
  - Index 8 requires offset of 32
    - 4 bytes per word

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



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
```


General Purpose Registers (GPRs) Dominate

° 1975-1995 all machines use general purpose registers

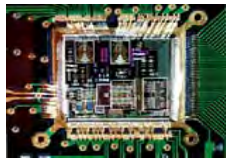
° Advantages of registers
  - registers are faster than memory
  - registers are easier for a compiler to use
  - registers can hold variables
    - memory traffic is reduced, so program is speeded up (since registers are faster than memory)
    - code density improves (since register named with fewer bits than memory location)

Registers vs. Memory

- In MIPS processor, arithmetic instructions operands must be registers.
- Registers are faster to access than memory
- Only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables?

  Solution: *Spilling Registers*

  Excessive variables are stored in Memory and moved from memory to register file by *load* and *store* instructions.



MIPS: Software conventions for Registers

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)

Plus a 3-deep stack of mode bits.

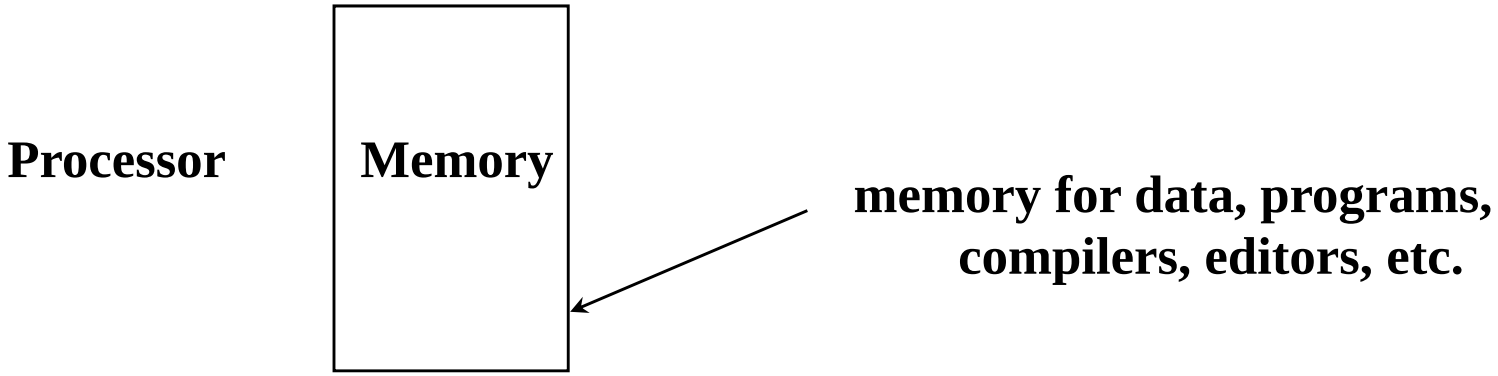
Architecture Styles …

According to the operand(s) locations..

- Accumulator-style: One of the operands is in an implicit register known as accumulator
- Load-store architecture: Both operands must be in the registers
- Register-memory: One operand in register, the other in Memory
- Memory-Memory: Both operands can be in Memory
- Stack-style: Stack is used to evaluate expressions

Stored Program Concept

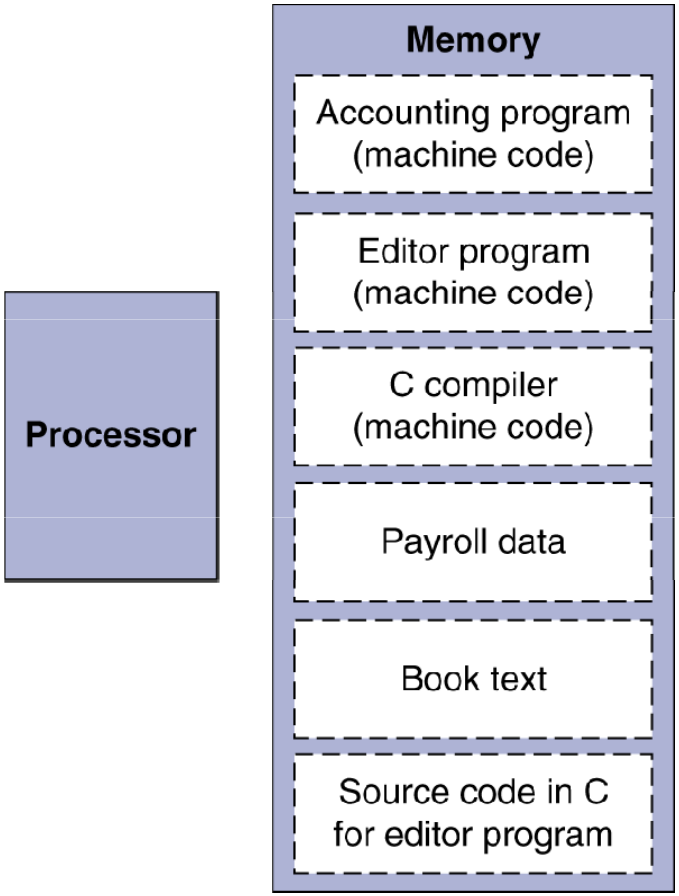
- Instructions are bits
- Programs are stored in memory
  — to be read or written just like data



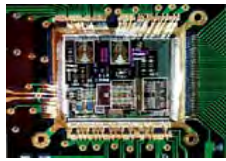
- Fetch & Execute Cycle
  - Instructions are fetched and put into a special register.
  - Bits in the register "control" the subsequent actions.
  - Fetch the “next” instruction and continue.

Stored Program Computers

The BIG Picture

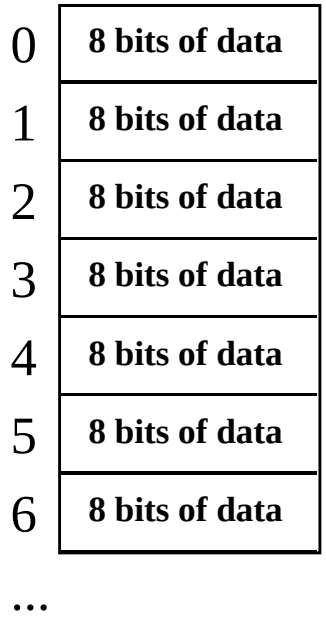


- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs



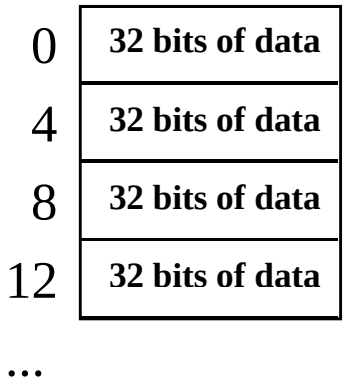
Memory Organization

- 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.



Memory Organization

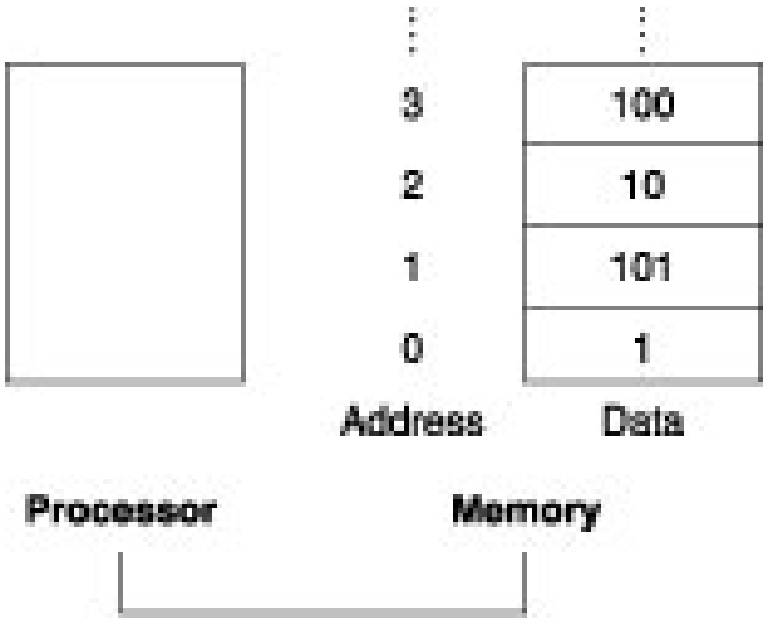
- Bytes are nice, but most data items use larger "words".
- For MIPS, a word is 32 bits or 4 bytes.



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?

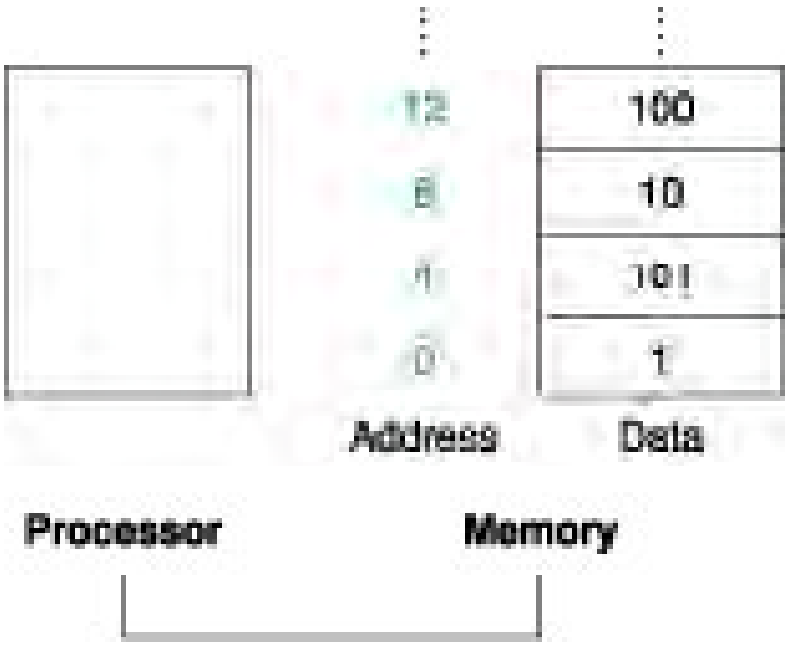
Memory Addresses and Contents



- Address of 3rd element is 2 and the value of Memory[2] is 10.

- Arithmetic operations occurs only on registers in MIPS.
- Data transfer instructions needed to transfer between memory and registers.
- Two types:
  - load word : from memory to register
  - store word : from register to memory

MIPS Memory Addresses and Contents



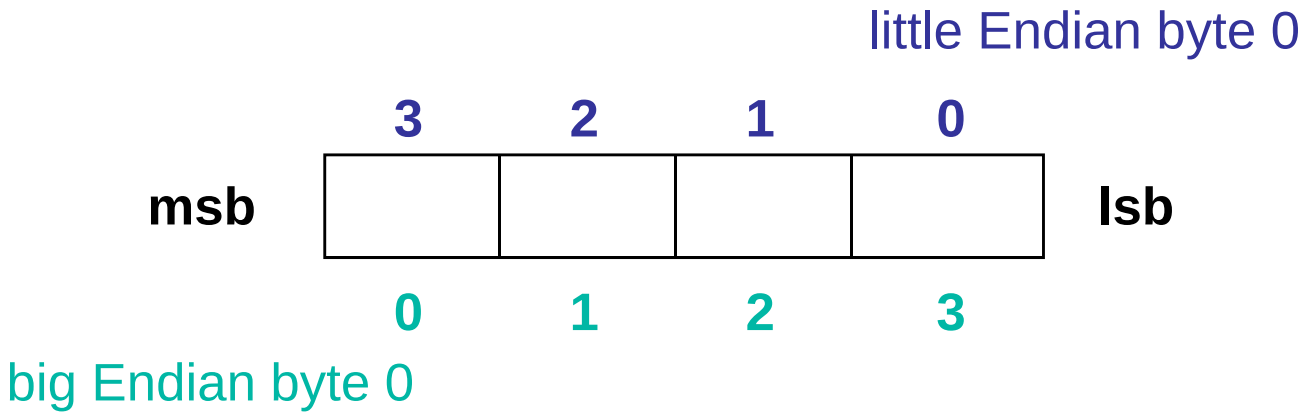
- Address of 3rd element is 8 and the value of Memory[8] is 10.
- Byte addressing in array: Base address + Offset.
- Offset = 4 * array index.

- In MIPS, word addresses start at multiple of 4.
- This is *alignment restriction*.

Addressing Objects: Endianess

- Big Endian: address of most significant byte
  IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

- Little Endian: address of least significant byte
  Intel 80x86, DEC Vax, DEC Alpha (Windows NT)



Constants or Immediate Operands

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

  e.g.,

  A = A + 15;

  B = B - 18;

  counter = counter + 1;

- In most programs, constants will fit in 16 bits allocated for immediate field.

Immediate Operands

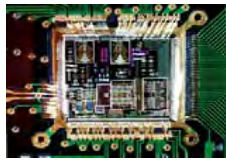
- Constant data specified in an instruction

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

- No subtract immediate instruction
  - Just use a negative constant

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

- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction



The Constant Zero

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

```
add $t2, $s1, $zero
```


Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: 0 to $+2^n - 1$
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
  - 0 to +4,294,967,295

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits
  - –2,147,483,648 to +2,147,483,647

2s-Complement Signed Integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- $-(-2^{n-1})$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

Signed Negation

- Complement and add 1
  - Complement means 1 → 0, 0 → 1

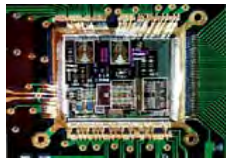
$$x + \bar{x} = 1111...111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Example: negate +2
  - +2 = 0000 0000 … 0010_2
  - –2 = 1111 1111 … 1101_2 + 1
    = 1111 1111 … 1110_2

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
- In MIPS instruction set
  - `addi`: extend immediate value
  - `lb`, `lh`: extend loaded byte/halfword
  - `beq`, `bne`: extend the displacement
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110



Load & Store Instructions by Example

- Load and store instructions are used for data movement between memory and registers in the *register file*.
- Example:

  C code: A[8] = h + A[8];

  MIPS code:
  ```
  lw $t0, 32($s3)    #  $t0 = A[8]
  add $t0, $s2, $t0  #  $t0 = h + $t0
  sw $t0, 32($s3)    #  A[8] = $t0
  ```

  Note: (1) lw = load word, sw = store word

  (2) $t0 is a temporary register that accumulates the final result

  (3) Register $s2 holds variable "h"

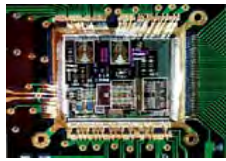
  (4) Register $s3 is the index register that holds the start address of the array A i.e. the location where array A starts.
- Store word has destination last
- Remember arithmetic operands are registers, not memory!

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



Conditional Operations

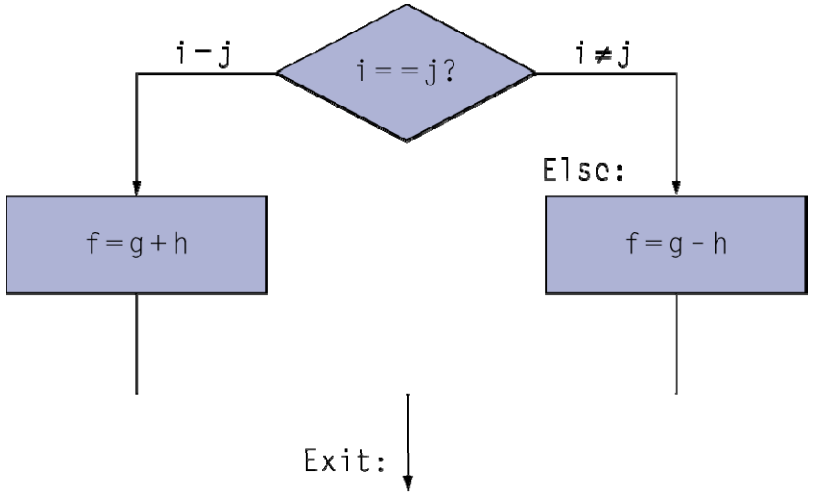
- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs, rt, L1`
  - if (rs == rt) branch to instruction labeled `L1`;
- `bne rs, rt, L1`
  - if (rs != rt) branch to instruction labeled `L1`;
- `j L1`
  - unconditional jump to instruction labeled `L1`

Compiling If Statements

- C code:

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

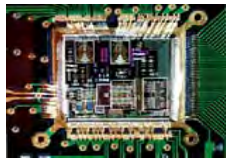
  - f, g, … in $s0, $s1, …



- Compiled MIPS code:

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

Assembler calculates addresses



Compiling Loop Statements

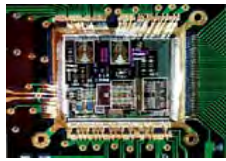
- C code:

```
while (save[i] == k) i += 1;
```

  - i in $s3, k in $s5, address of save in $s6

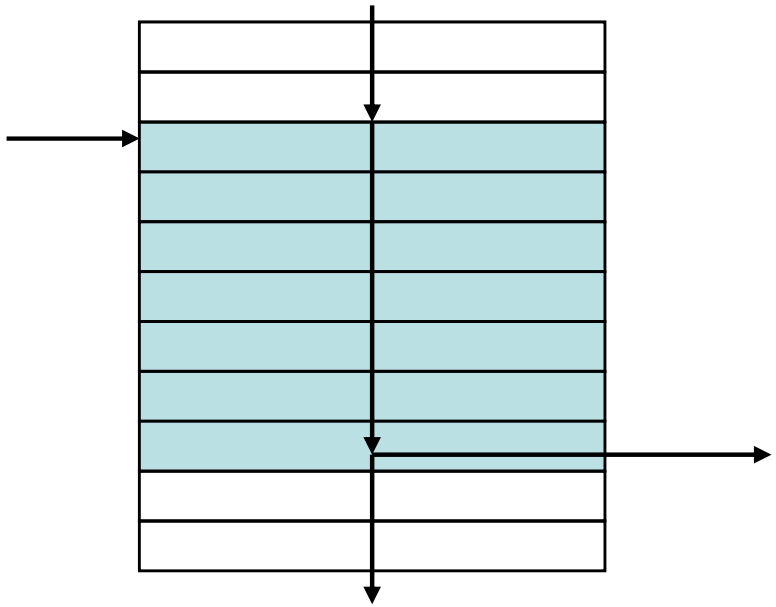
- Compiled MIPS code:

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

More Conditional Operations

- Set result to 1 if a condition is true
  - Otherwise, set to 0
- `slt rd, rs, rt`
  - if (rs < rt) rd = 1; else rd = 0;
- `slti rt, rs, constant`
  - if (rs < constant) rt = 1; else rt = 0;
- Use in combination with `beq`, `bne`

```
slt $t0, $s1, $s2  # if ($s1 < $s2)
bne $t0, $zero, L  #   branch to L
```


Branch Instruction Design

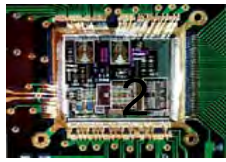
- Why not `blt`, `bge`, etc?
- Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
- `beq` and `bne` are the common case
- This is a good design compromise

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
- Note that the assembler needs a register to do this,
  — there are policy of use conventions for registers



Signed vs. Unsigned

- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`
- Example
  - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `slt  $t0, $s0, $s1  # signed`
    - –1 < +1 ⇒ $t0 = 1
  - `sltu $t0, $s0, $s1  # unsigned`
    - +4,294,967,295 > +1 ⇒ $t0 = 0

Instructions for Control flow

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

- MIPS conditional branch instructions:

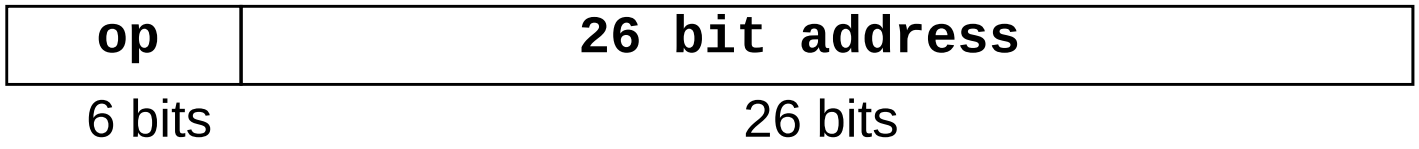
```
bne $t0, $t1, Label
beq $t0, $t1, Label
```

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

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


Unconditional Branch: jump instruction

- MIPS unconditional branch instructions:
  j label
- Jump Instruction Format:



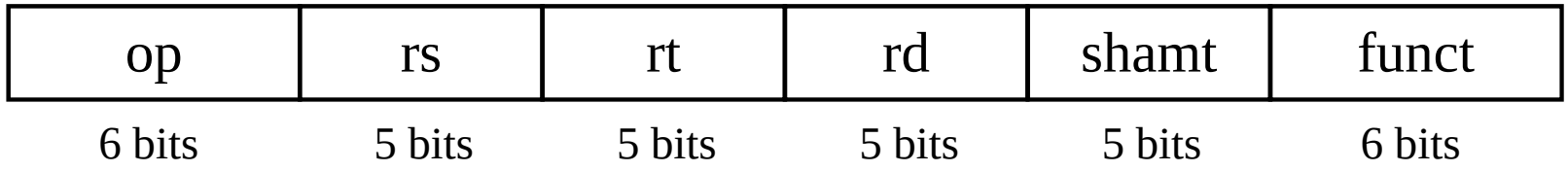
- Example:

```
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:  ...
```

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
  - $t0 – $t7 are reg's 8 – 15
  - $t8 – $t9 are reg's 24 – 25
  - $s0 – $s7 are reg's 16 – 23

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)

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

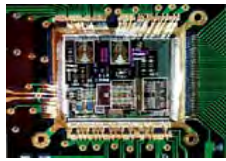
$$00000010001100100100000000100000_2 = 02324020_{16}$$

Hexadecimal

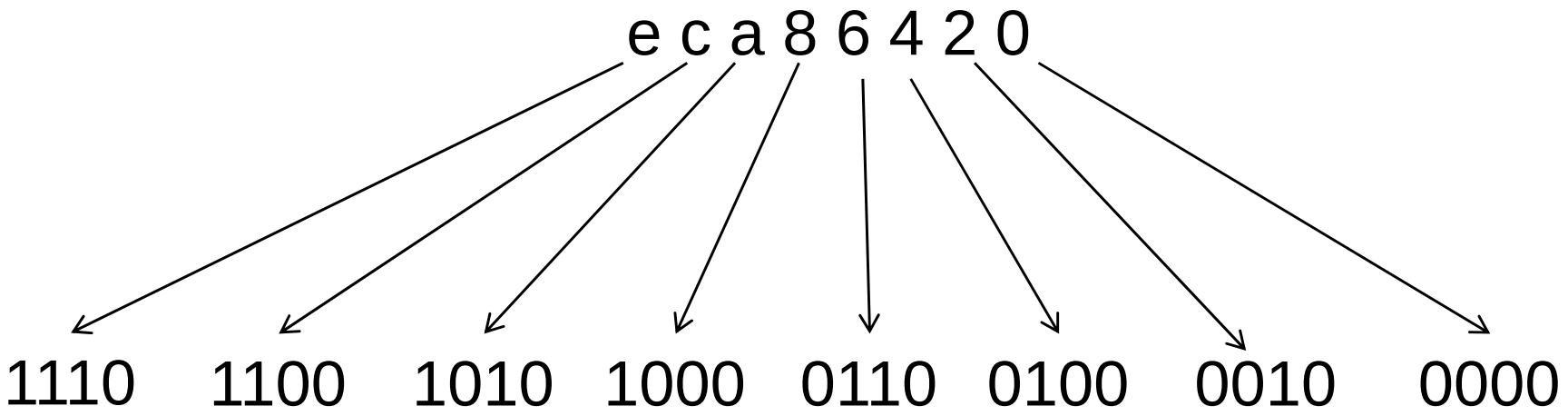
- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

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

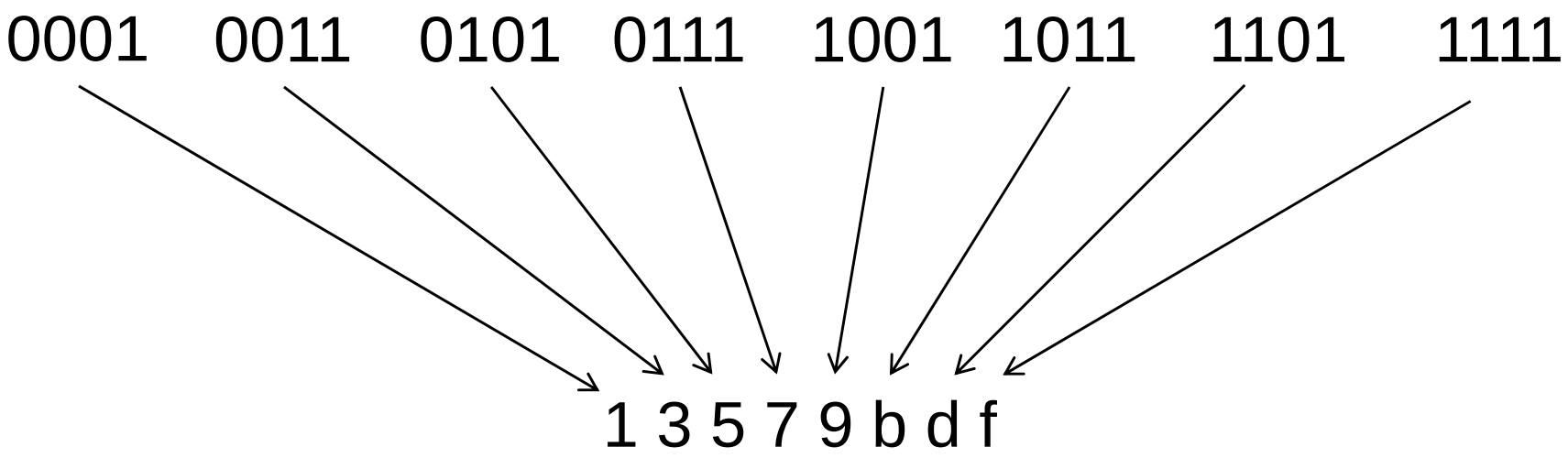
- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000



Data Types: Binary to Hexadecimal



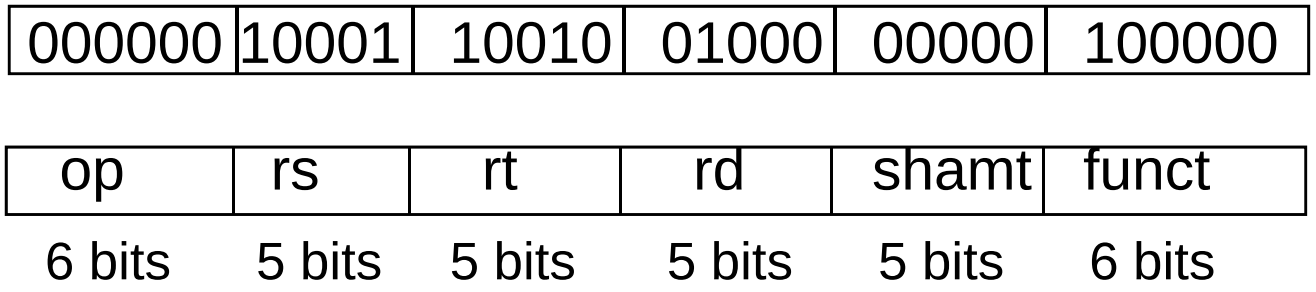
Data Types: Hexadecimal to Binary



Machine Language

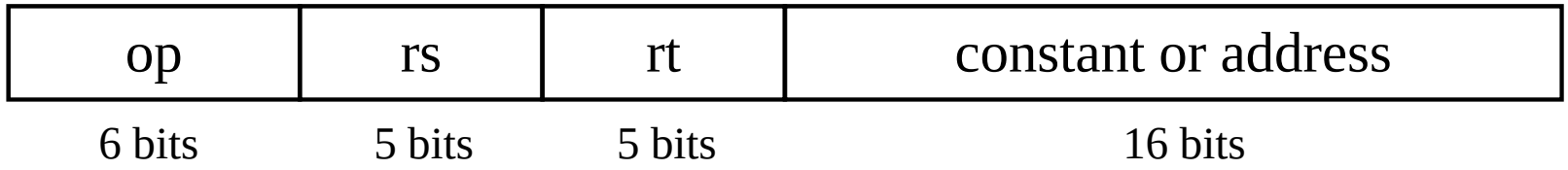
- Instructions, like registers and words of data, are also 32 bits long
  - Example: add $t0, $s1, $s2
  - registers have numbers, t0=8, s1=17, s2=18

- Instruction Format (R-type):



op: operation of the instruction
rs: the first register source operand
rt: the second register source operand
shamt: shift amount (we will look at this later..)
funct: function; this field selects the variant of the operation in the op field

MIPS I-format Instructions



- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15} - 1$
  - Address: offset added to base address in rs
- *Design Principle 4:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

Machine Language

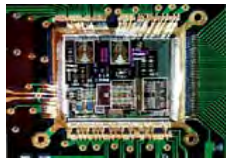
- Consider the load-word and store-word instructions,
  - What would the regularity principle have us do?
  - New principle: Good design demands a compromise

- 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)

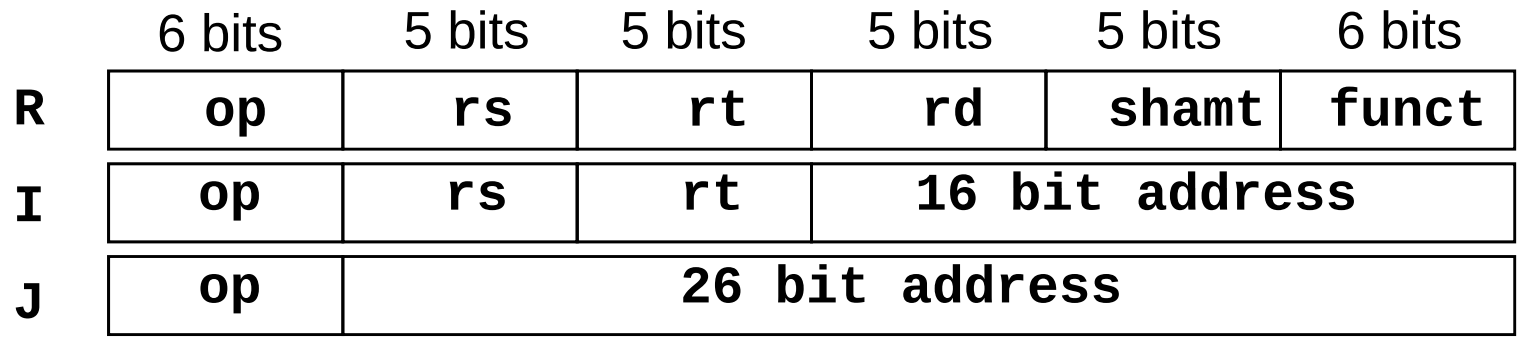
35	18	9	32
6 bits	5 bits	5 bits	16 bits
op	rs	rt	16 bit number

- Where's the compromise?

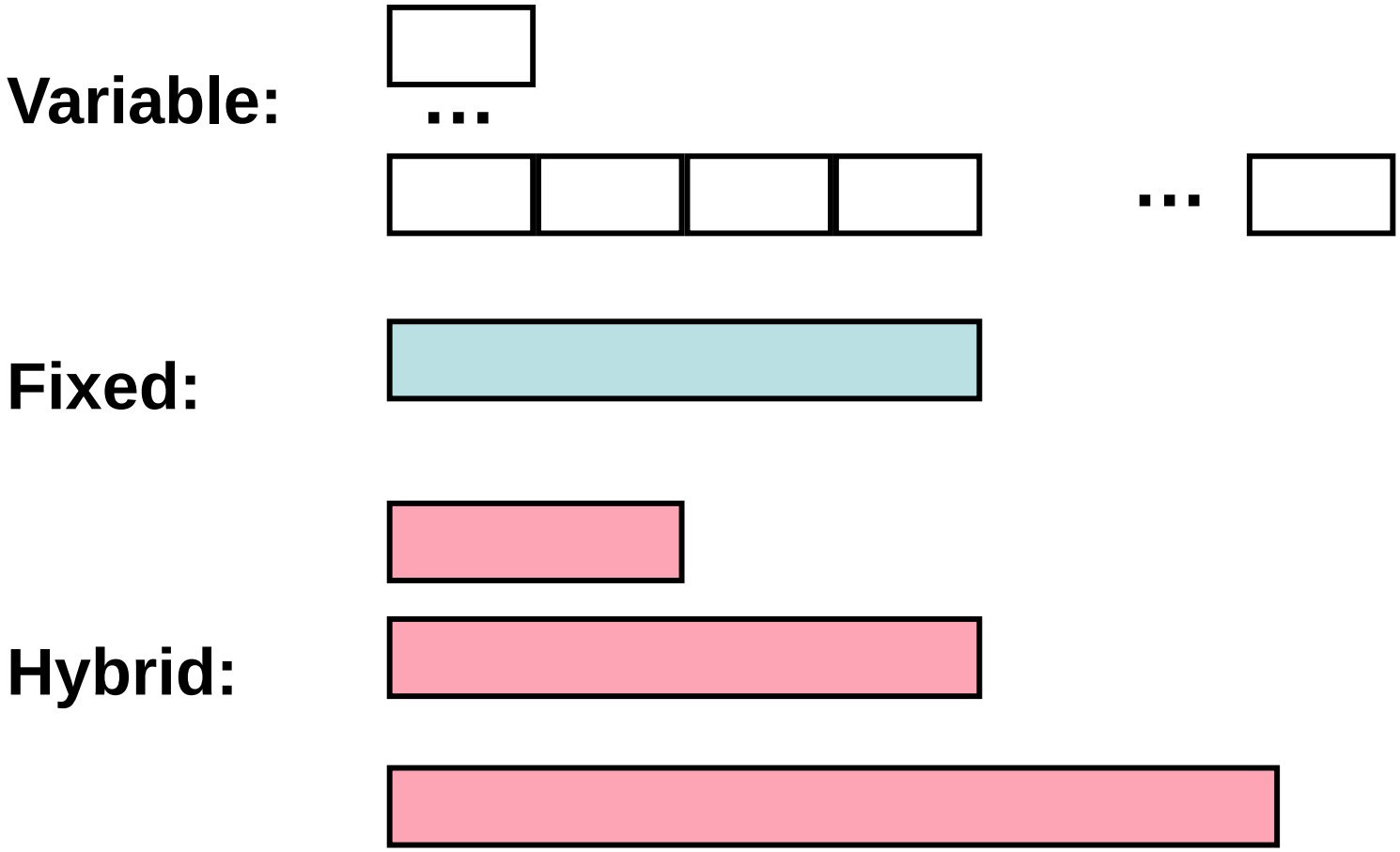


MIPS Instructions: So far …

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,L	Next instr. is at Label if $s4 != $s5
beq $s4,$s5,L	Next instr. is at Label if $s4 = $s5
j Label	Next instr. is at Label
- Formats:



Generic Examples of Instruction Format Widths



Summary of Instruction Formats

- If code size is most important, use variable length instructions.
- If performance is most important, use fixed length instructions.
- Recent embedded machines (ARM, MIPS) added optional mode to execute subset of 16-bit wide instructions (Thumb, MIPS16); per procedure decide performance or density.

Assembly versus Machine Language

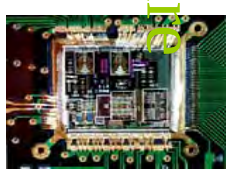
- Assembly provides convenient symbolic representation
  - much easier than writing down numbers
  - e.g., destination first
- Machine language is the underlying reality
  - e.g., destination is no longer first
- Assembly can provide *'pseudoinstructions*'
  - e.g., “move $t0, $t1” exists only in Assembly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

Other Issues

- Several other issues can be considered:
  - support for procedures
  - linkers, loaders, memory layout
  - stacks, frames, recursion
  - manipulating strings and pointers
  - interrupts and exceptions
  - system calls and conventions
- We've focused on architectural issues
  - basics of MIPS assembly language and machine code.
  - we'll build a processor to execute these instructions.

Procedure Calling

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



Register Usage

- $a0 – $a3: arguments (reg's 4 – 7)
- $v0, $v1: result values (reg's 2 and 3)
- $t0 – $t9: temporaries
  - Can be overwritten by callee
- $s0 – $s7: saved
  - Must be saved/restored by callee
- $gp: global pointer for static data (reg 28)
- $sp: stack pointer (reg 29)
- $fp: frame pointer (reg 30)
- $ra: return address (reg 31)

Procedure Call Instructions

- Procedure call: jump and link

  `jal ProcedureLabel`

  – Address of following instruction put in $ra

  – Jumps to target address

- Procedure return: jump register

  `jr $ra`

  – Copies $ra to program counter

  – Can also be used for computed jumps

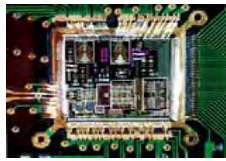
    - e.g., for case/switch statements

Leaf Procedure Example

- C code:

```
int leaf_example (int g, h, i, j)
{
   int f;
   f = (g + h) - (i + j);
   return f;
}
```

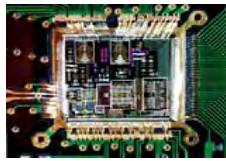
  - Arguments g, …, j in $a0, …, $a3
  - f in $s0 (hence, need to save $s0 on stack)
  - Result in $v0



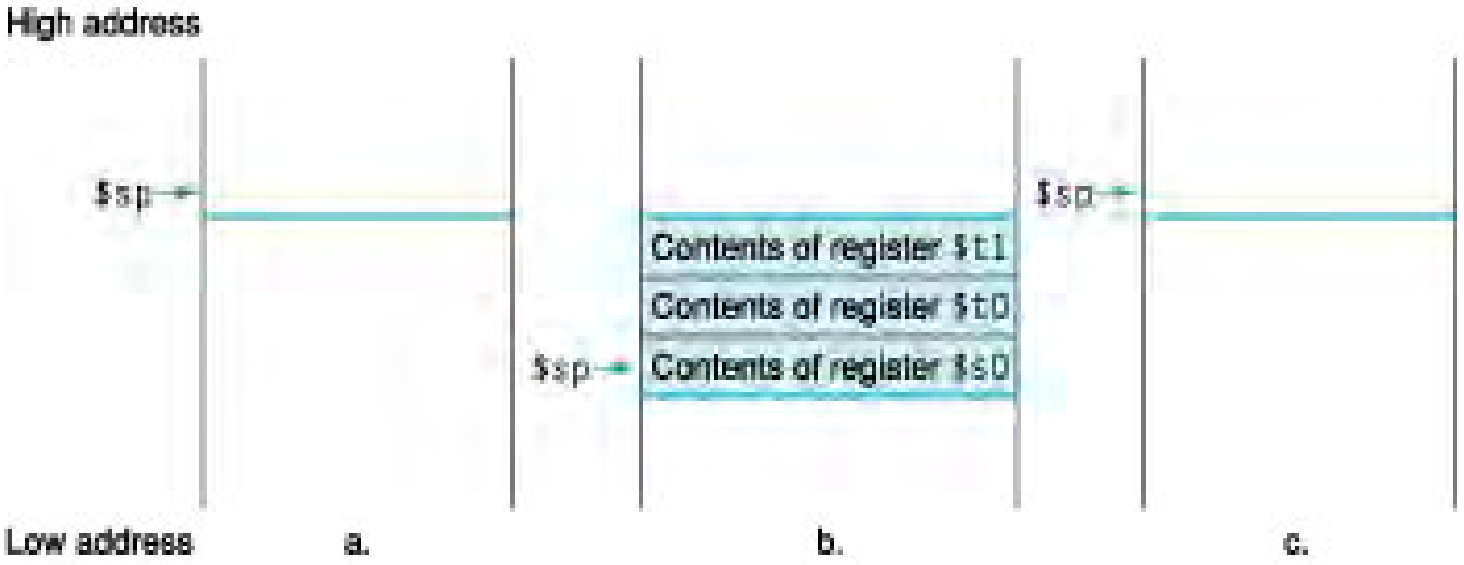
Leaf Procedure Example

- MIPS code:

```
leaf_example:
        addi $sp, $sp, -12      # Make room is stack for 3 item
        sw   $t1, 8($sp)
        sw   $t0, 4($sp)        # Save $t1, $t0, $s0 on stack
        sw   $s0, 0($sp)
        add  $t0, $a0, $a1
        add  $t1, $a2, $a3      # Procedure body
        sub  $s0, $t0, $t1
        add  $v0, $s0, $zero    # Result, Return value of f
        lw   $s0, 0($sp)
        lw   $t0, 4($sp)        # Restore $t1, $t0, $s0
        lw   $t1, 8($sp)
        addi $sp, $sp, 12       # Adjust stack to delete 3 item
        jr   $ra                # Return to the calling routine
```

Stack Pointer

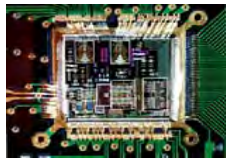


Before During After

- Values of stack pointer and stack for procedure call.
- The stack pointer always points to the “top” of the stack, or the last word in the stack in this drawing.

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

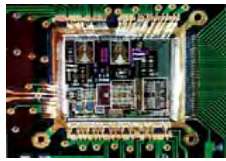


Non-Leaf Procedure Example

- C code:

```
int fact (int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

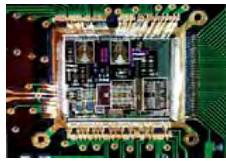
  - Argument n in $a0
  - Result in $v0



Non-Leaf Procedure Example

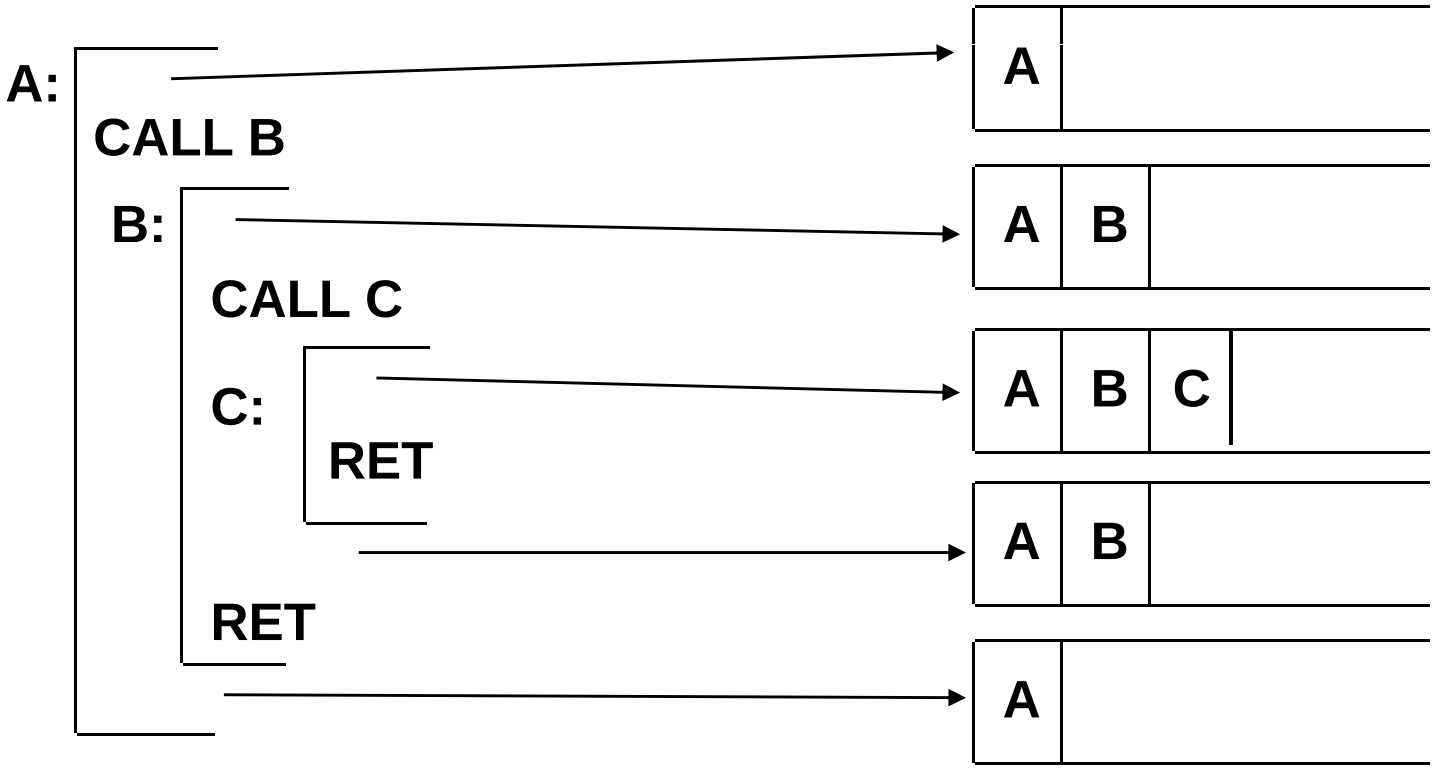
- MIPS code:

```
fact:
    addi $sp, $sp, -8       # adjust stack for 2 items
    sw   $ra, 4($sp)        # save return address
    sw   $a0, 0($sp)        # save argument
    slti $t0, $a0, 1        # test for n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1      # if so, result is 1
    addi $sp, $sp, 8        #   pop 2 items from stack
    jr   $ra                #   and return
L1: addi $a0, $a0, -1       # else decrement n
    jal  fact               # recursive call
    lw   $a0, 0($sp)        # restore original n
    lw   $ra, 4($sp)        #   and return address
    addi $sp, $sp, 8        # pop 2 items from stack
    mul  $v0, $a0, $v0      # multiply to get result
    jr   $ra                # and return to the caller
```

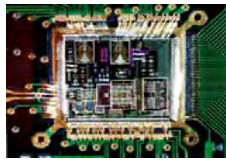



Calls: Why Are Stacks So Great?

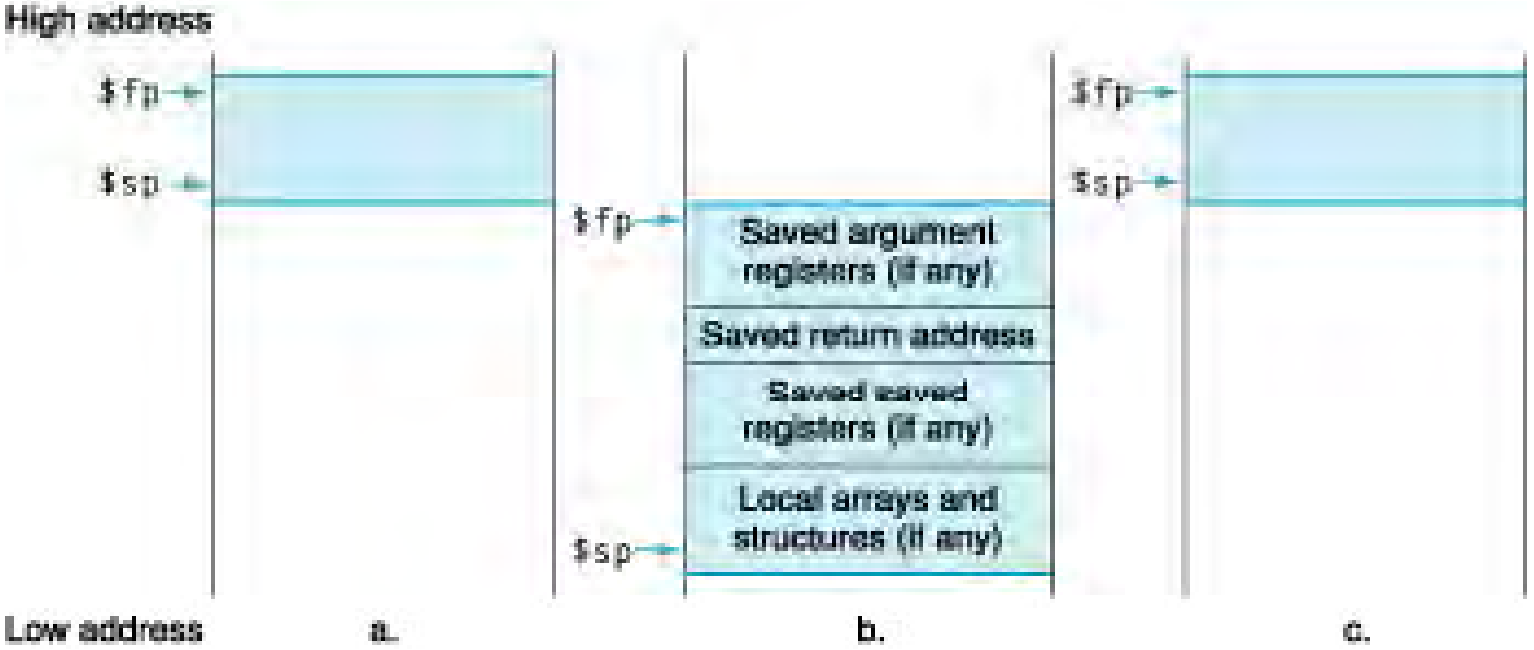
Stacking of Subroutine Calls & Returns and Environments:



- Some machines provide a memory stack as part of the architecture (e.g., VAX)
- Sometimes stacks are implemented via software convention (e.g., MIPS)



Frame and Stack Pointer

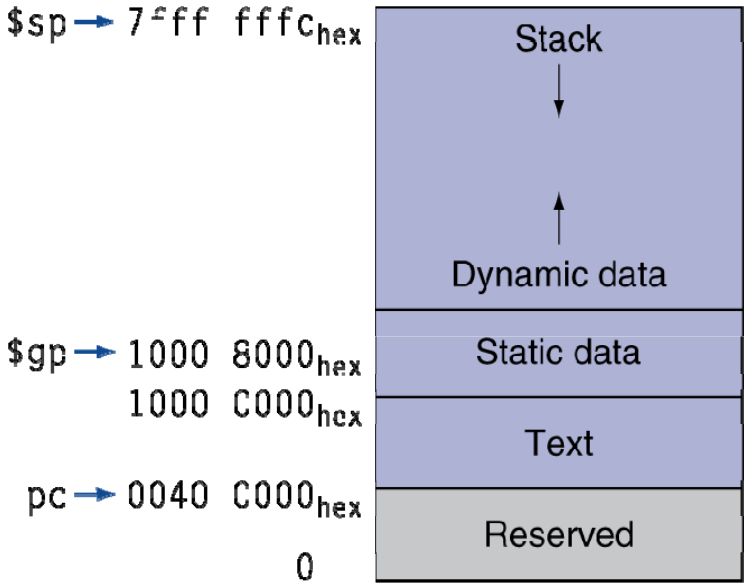


Before During After

- The frame pointer points to the first word in the frame of a procedure.
- Frame pointer is a saved argument register.
- The stack is adjusted to make room for all saved registers and any memory-resident local variables.

Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: Heap
  - e.g., malloc in C, new in Java
- Stack: automatic storage



Overview of MIPS

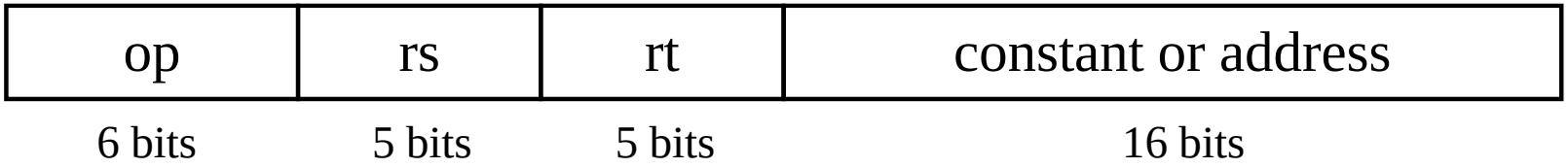
- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
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.

Branch Addressing

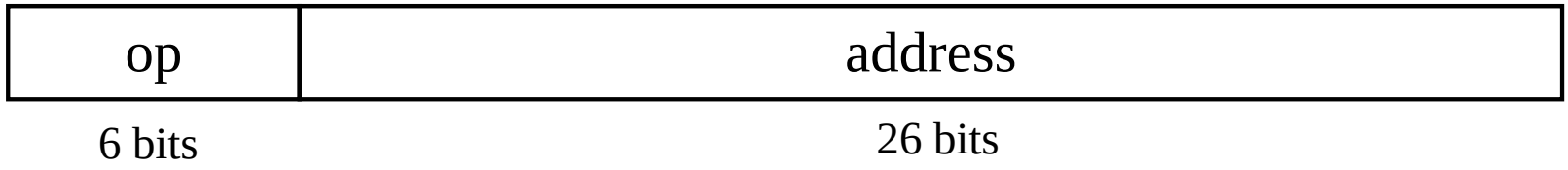
- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward



- PC-relative addressing
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time

Jump Addressing

- Jump (`j` and `jal`) targets could be anywhere in text segment
  - Encode full address in instruction



- (Pseudo)Direct jump addressing
  - Target address = $PC_{31\ldots28}$: (address × 4)

Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

```
Loop: sll  $t1, $s3, 2      80000
      add  $t1, $t1, $s6    80004
      lw   $t0, 0($t1)      80008
      bne  $t0, $s5, Exit   80012
      addi $s3, $s3, 1      80016
      j    Loop             80020
Exit: …                     80024
```

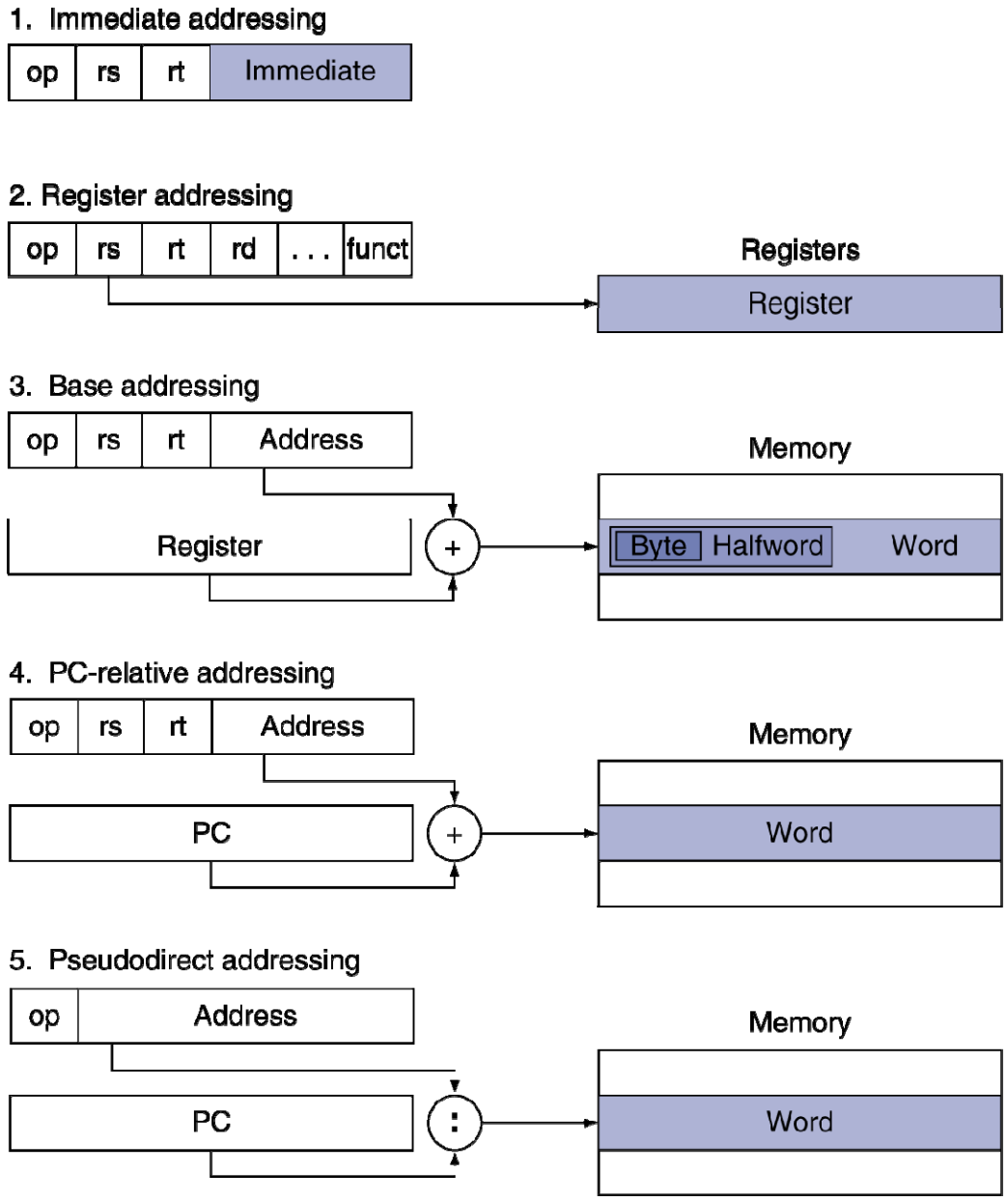
0	0	19	9	4	0
0	9	22	9	0	32
35	9	8	0		
5	8	21	2		
8	19	19	1		
2	20000				

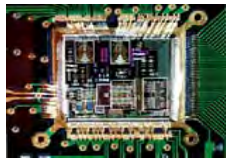
Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

```
        beq $s0,$s1, L1
                ↓
        bne $s0,$s1, L2
        j L1
L2:     …
```


Addressing Mode Summary





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
Uncondi-tional 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