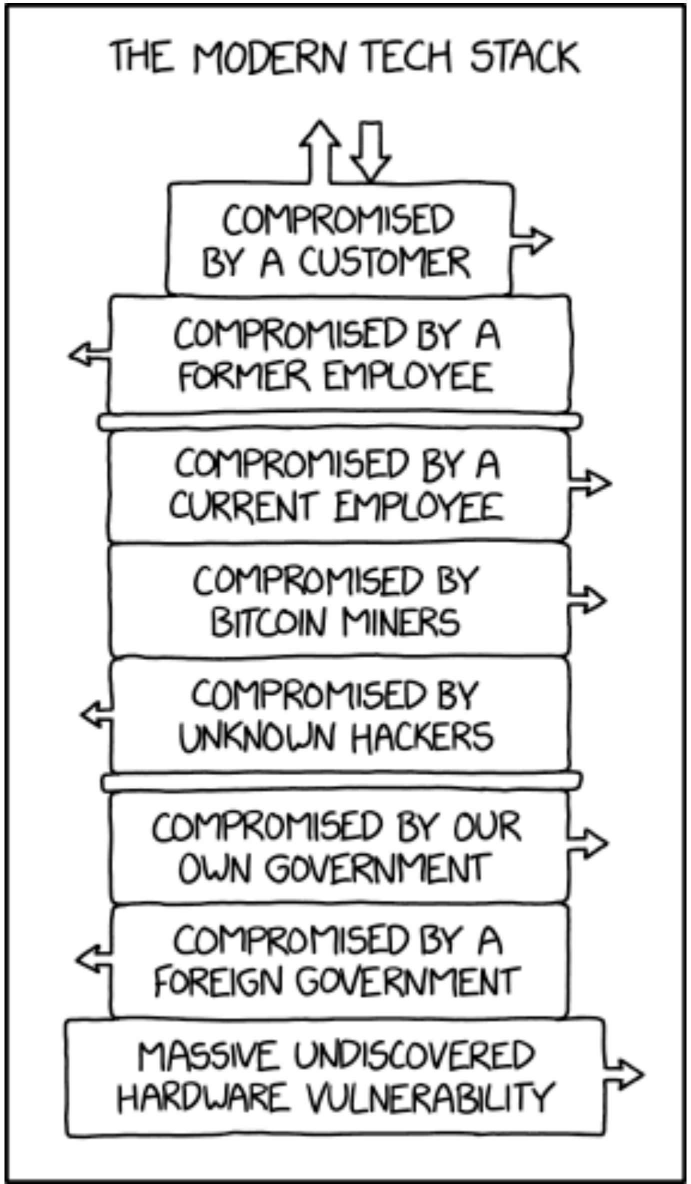

THE MODERN TECH STACK
COMPROMISED BY A CUSTOMER
COMPROMISED BY A FORMER EMPLOYEE
COMPROMISED BY A CURRENT EMPLOYEE
COMPROMISED BY BITCOIN MINERS
COMPROMISED BY UNKNOWN HACKERS
COMPROMISED BY OUR OWN GOVERNMENT
COMPROMISED BY A FOREIGN GOVERNMENT
MASSIVE UNDISCOVERED HARDWARE VULNERABILITY

Instruction Set

- Different computers have different instruction sets
  - But with many aspects in common

- You might think that the languages of computers would be as diverse as those of people, but in reality computer languages are quite similar, more like regional dialects than like independent languages.

- The chosen instruction set comes from MIPS Technologies, and is an elegant example of the instruction sets designed since the 1980s.

Instruction Set

- ARMv7 is similar to MIPS. More than 9 billion chips with ARM processors were manufactured in 2011
- Intel x86, which powers both the PC and the cloud of the PostPC Era.
- ARMv8, which extends the address size of the ARMv7 from 32 bits to 64 bits. Ironically, this 2013 instruction set is closer to MIPS than it is to ARMv7

Stored-program concept

- Storage of instructions in computer memory to enable it to perform a variety of tasks in sequence or intermittently
- Created by Jon von Neumann in the late 1940s.
- Manchester Mark 1 – proposed and implemented in England
- EDVAC -- proposed and implemented in the US.

Operations of the Computer Hardware

- The MIPS assembly language notation

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

instructs a computer to add the two variables `b` and `c` and to put their sum in `a`

- MIPS arithmetic instruction performs only one operation and must always have exactly three variables.

MIPS Example:

- For example, suppose we want to place the sum of four variables `b`, `c`, `d`, and `e` into variable `a`.

```
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
```

every newline is a new instruction. No ;

"#" is how to comment

Example: C → MIPS

- This segment of a C program contains the five variables `a, b, c, d, and e`

```
a = b + c;
d = a − e;
```

- A MIPS instruction operates on two source operands and places the result in one destination operand.

MIPs Instruction Goals:

- Showing how it is represented in hardware
- The relationship between high-level programming languages and this more primitive one
  - Examples in C and JAVA
- See the impact of programming languages and compiler optimization on performance
- writing programs in the language of the computer and running them on the simulator

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

Example: C → MIPS

- A somewhat complex statement contains the five variables `f, g, h, i, and j`:

```
f = (g + h) – (i + j);
```

➢Need to use temporary variables

➢Break it into 3 different MIPS instructions

Register Operand Example

- C code:

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

  ASSUME: f, g, h, i, and j are assigned to the registers $s0, $s1, $s2, $s3, and $s4, respectively. Compiled MIPS code:

```
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)
```

Data Types

- Instructions are all 32 bits
- byte(8 bits), halfword (2 bytes), word (4 bytes)
- a character requires 1 byte of storage
- an integer requires 1 word (4 bytes) of storage

Immediate Operands

- Constant data specified in an instruction

```
addi $s3, $s3, 4 # add immediate $s3 = $s3 + 4
```

- No subtract immediate instruction
  - Just use a negative constant

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

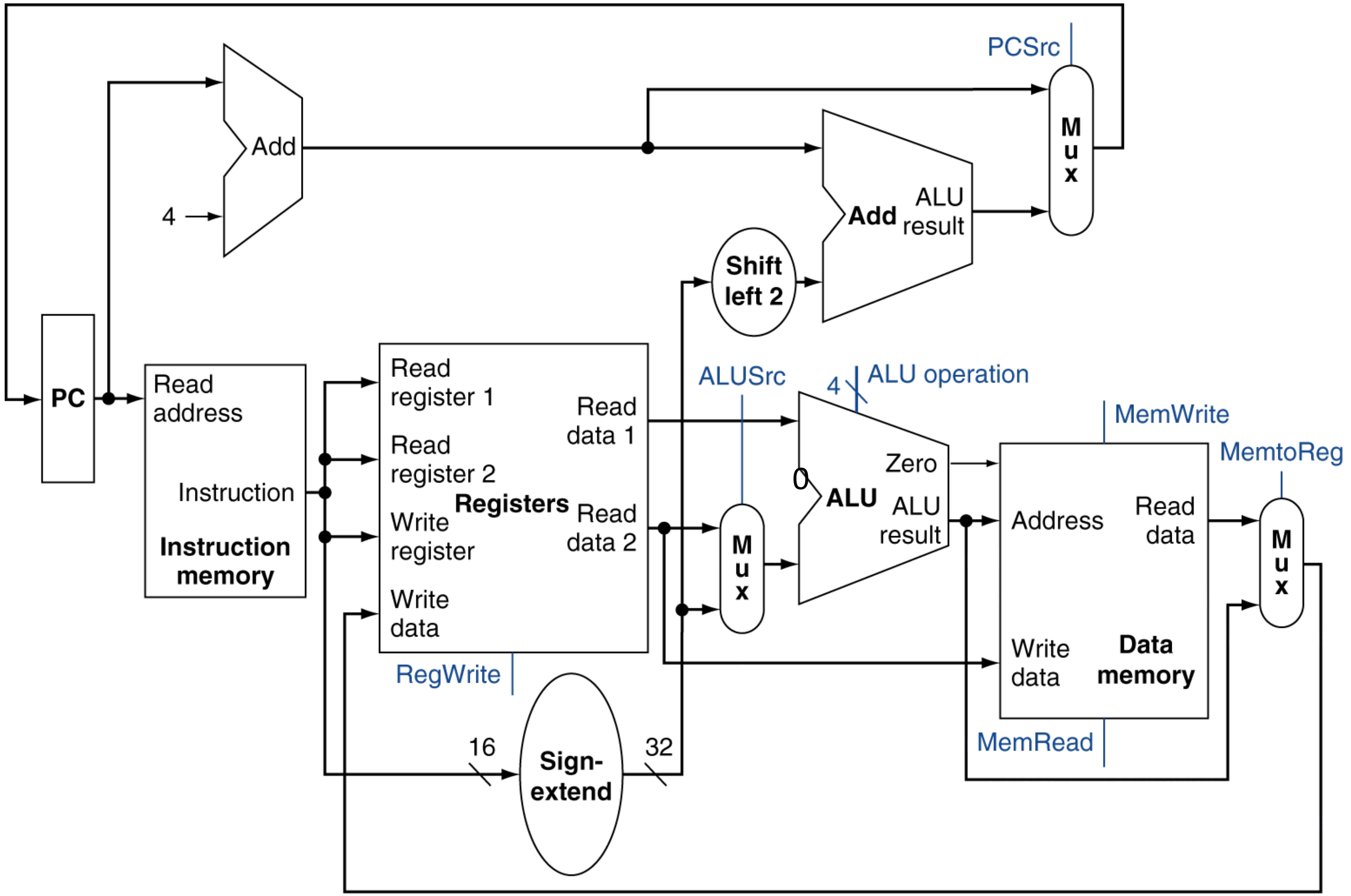
- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

The Constant Zero Register

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

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

MIPS Architecture

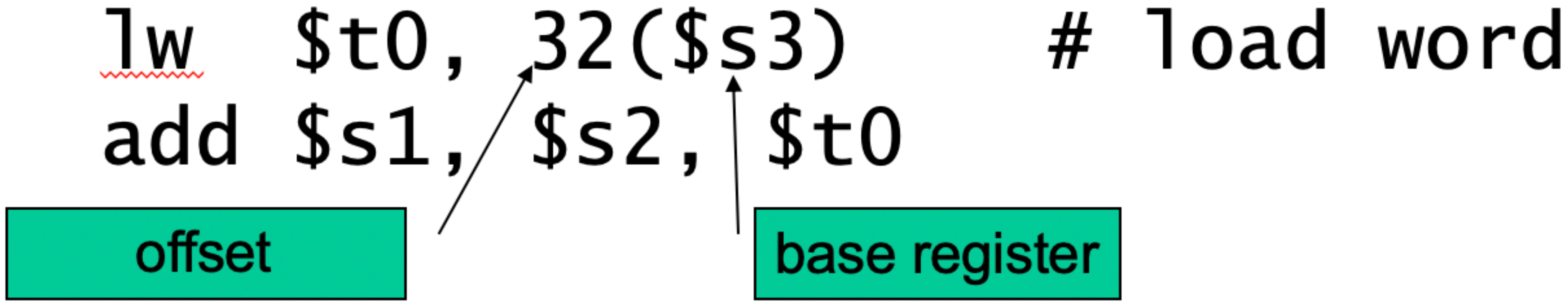


Memory Operand

To apply arithmetic operations

- Load values from memory into registers
- Store result from register to memory

Memory Operand



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

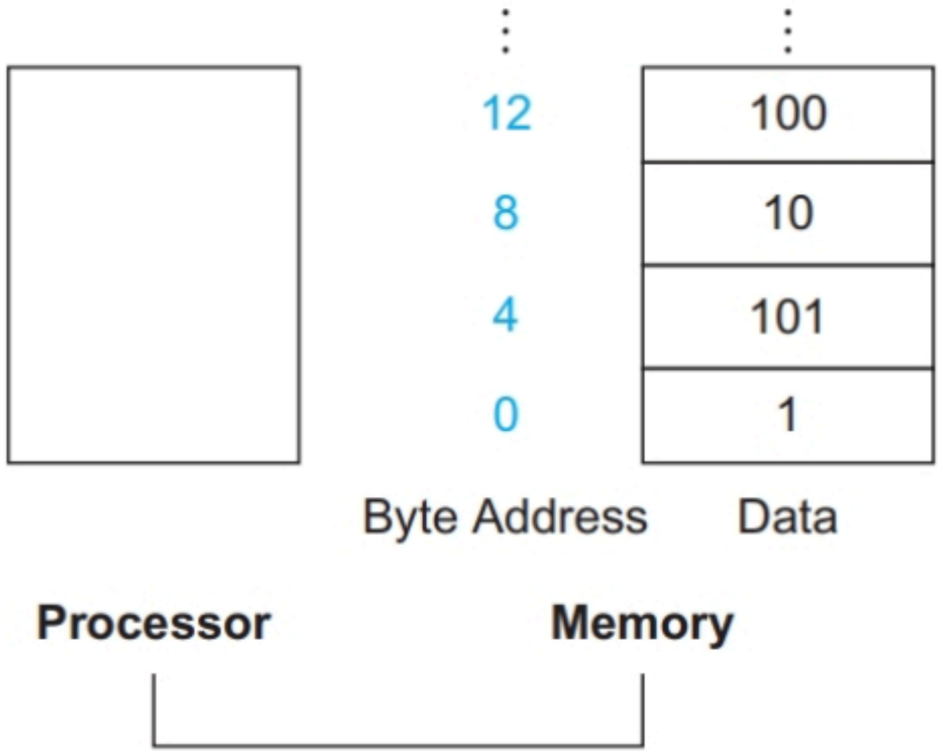
Memory Operand

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

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

- Index 8 requires offset of 32

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



Memory Operand Example

assuming h → S2 and Base address of A → S3

```
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]
```

$A[12] = h + A[8]$

Register Number	**Alternative Name**	**Description**
0	zero	the value 0
1	$at	(**a**ssembler **t**emporary) reserved by the assembler
2-3	$v0 - $v1	(**v**alues) from expression evaluation and function results
4-7	$a0 - $a3	(**a**rguments) First four parameters for subroutine. Not preserved across procedure calls
8-15	$t0 - $t7	(**t**emporaries) Caller saved if needed. Subroutines can use w/out saving. Not preserved across procedure calls
16-23	$s0 - $s7	(**s**aved values) - Callee saved. A subroutine using one of these must save original and restore it before exiting. Preserved across procedure calls
24-25	$t8 - $t9	(**t**emporaries) Caller saved if needed. Subroutines can use w/out saving. These are in addition to $t0 - $t7 above. Not preserved across procedure calls.
26-27	$k0 - $k1	reserved for use by the interrupt/trap handler
28	$gp	**g**lobal **p**ointer. Points to the middle of the 64K block of memory in the static data segment.
29	$sp	**s**tack **p**ointer Points to last location on the stack.
30	$s8/$fp	**s**aved value / **f**rame **p**ointer Preserved across procedure calls
31	$ra	**r**eturn **a**ddress

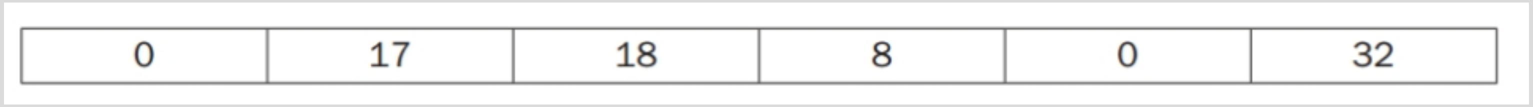
MIPS Instruction Computer Representation

- Instructions are kept in the computer as a series of high and low electronic signals
- May be represented as numbers
- Each piece of an instruction can be considered as an individual number
- Placing these numbers side by side forms the instruction.

MIPS Instruction Computer Representation

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

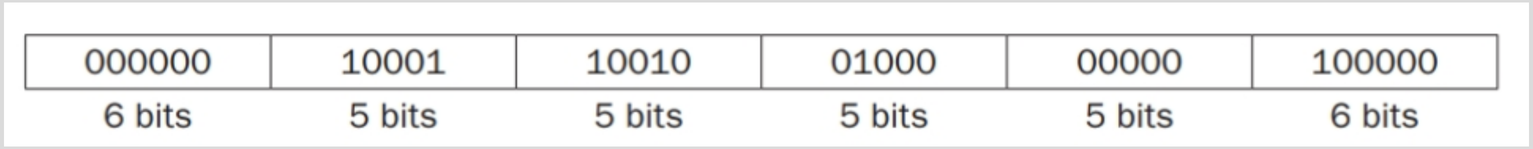
Decimal:



0	17	18	8	0	32

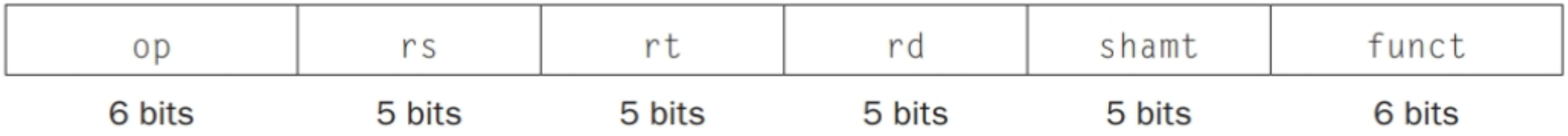
Field

Binary:



000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Instruction Format – Machine Language



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

op: Basic operation of the instruction, traditionally called the opcode.

rs: The first register source operand

rt: The second register source operand.

MIPS fields Defined

MIPS fields Defined

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

rd: The register destination operand. It gets the result of the operation

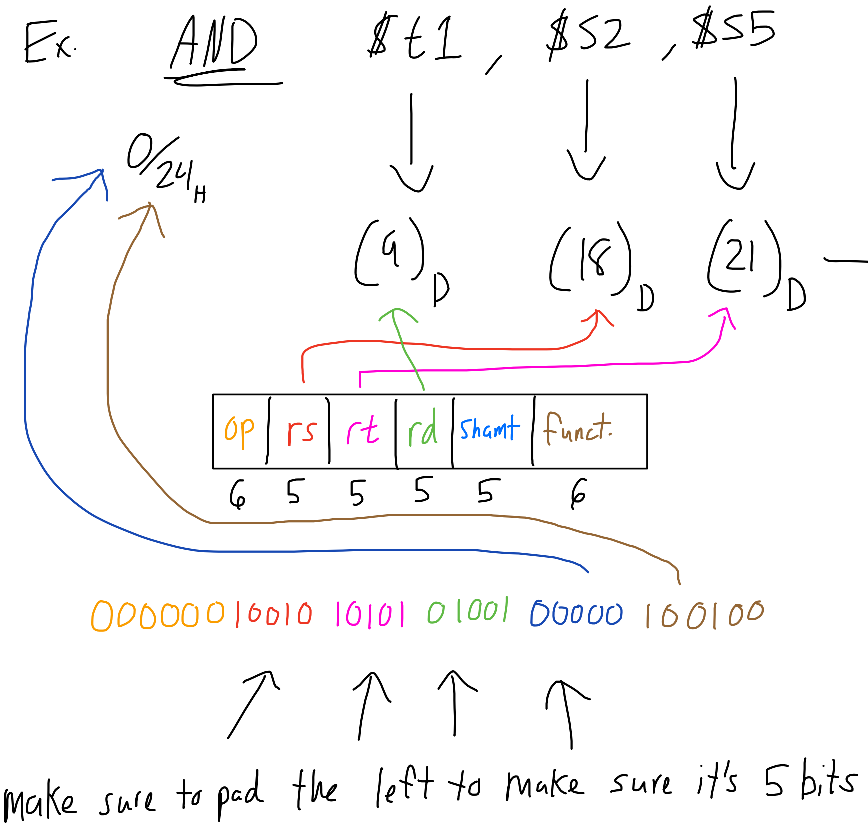
shamt: Shift amount

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

MIPS Instructions

- keep all instructions the same length
- Require different kinds of instruction formats for different kinds of instructions.
- R-format: r for registers (R-type)
- I-format: i for immediate (I-type)
- The formats are distinguished by the values in the first field each format is assigned a distinct set of values in the first field (op)
- the hardware knows whether to treat the last half of the instruction as three fields (R-type) or as a single field (I-type)

Ex. AND $t1, $S2, $S5



NAME	NUMBER
$zero	0
$at	1
$v0-$v1	2-3
$a0-$a3	4-7
$t0-$t7	8-15
$s0-$s7	16-23
$t8-$t9	24-25
$k0-$k1	26-27
$gp	28
$sp	29
$fp	30
$ra	31

0010 = 2
0100 = 4
But we only have 6 Bits
so we chop the leading zeros on 2
$24_H = 100100$

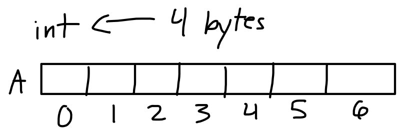
BTW: The largest num you can have in the function section is 3 F because it's 6 bits
11 1111

Slides 17-20

Ex. lw $t0, 32 ($s3)

The parentheses tell me that I want the value at that address and not the address

Base memory address



A[7]

I want to access A[0]

assume base address A is in $S2
lw $t1, 0 ($S2)

Base address

This loads the value of S2 into t1 and not the address

34 mins

Ex.

Take that yellow line of code and get the mips instructions for it

A[300] = h+A[300];

$t1 → Base of A $S2 → h

```
lw  $t0, 1200($t1)   # $t0 = A[300]
add $t0, $S2, $t0    # h + A[300]
sw  $t0, 1200($t1)   # A[300] = $t0
```

now I'm writing out the Binary sequence for this

This is a new question

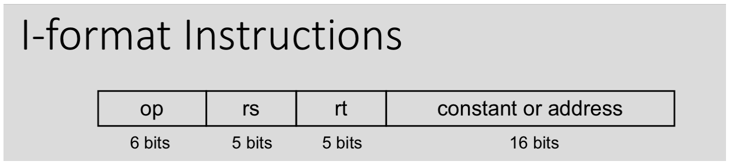
op = 23_{Hex}

rs = 9

rt = 8

Const = 1200

Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}



I-format Instructions

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

010011	01001	01000	0000010010110000
op	9	8	1200
	rs	rt	const

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

$00000010001100100100000000100000_2 = 02324020_{16}$

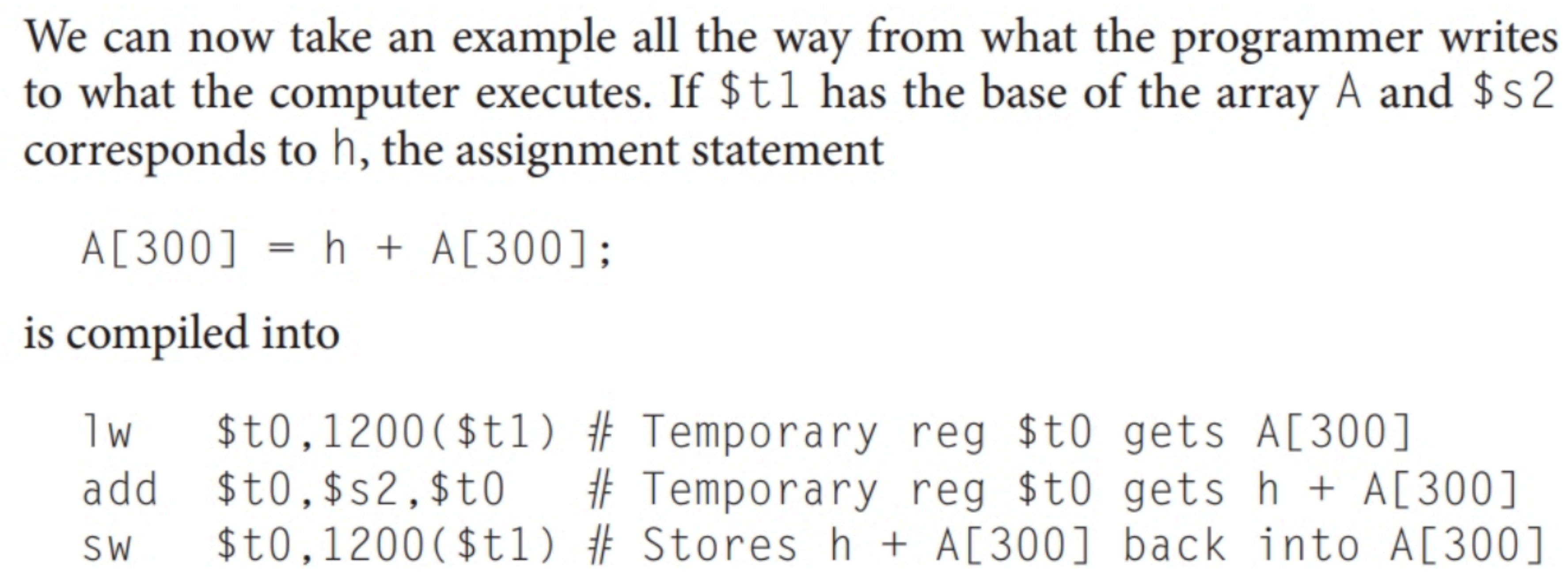
I-format Instructions

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- 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

Instruction	Format	op	rs	rt	rd	shamt	funct	address
`add`	R	0	reg	reg	reg	0	32_{ten}	n.a.
`sub` (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
`add immediate`	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
`lw` (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
`sw` (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

Example:



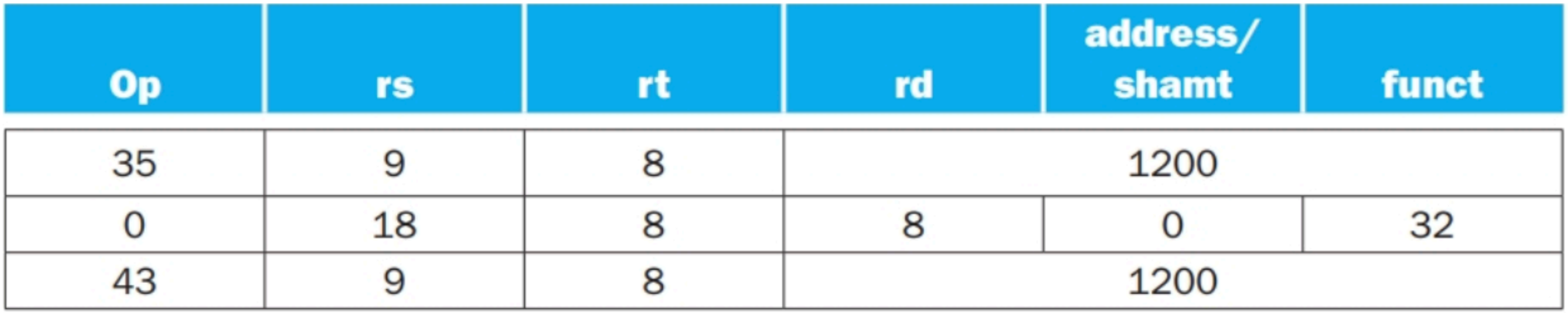
We can now take an example all the way from what the programmer writes to what the computer executes. If `$t1` has the base of the array `A` and `$s2` corresponds to `h`, the assignment statement

```
A[300] = h + A[300];
```

is compiled into

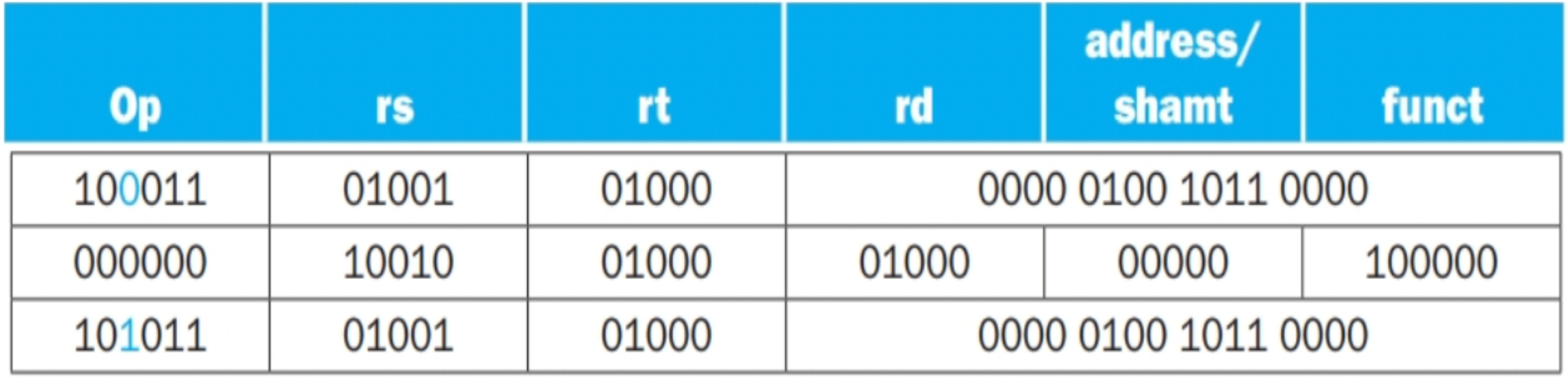
```
lw   $t0,1200($t1) # Temporary reg $t0 gets A[300]
add  $t0,$s2,$t0   # Temporary reg $t0 gets h + A[300]
sw   $t0,1200($t1) # Stores h + A[300] back into A[300]
```

Example: Machine Language Decimal Solution



Op	rs	rt	rd	address/ shamt	funct
35	9	8		1200	
0	18	8	8	0	32
43	9	8		1200	

Example: Machine Language Binary Solution



Op	rs	rt	rd	address/ shamt	funct
100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

Sign Extension

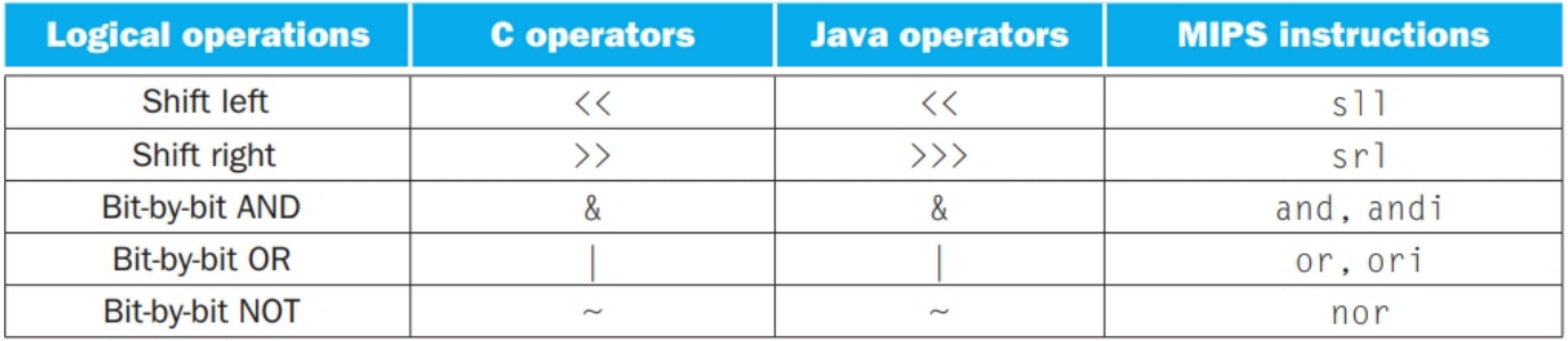
- Representing a number using more bits
  - Preserve the numeric value
- When loading a 32-bit word into a 32-bit register, signed and unsigned loads are identical.
- MIPS does offer two flavors of byte loads:
  - ***load byte (lb):*** treats the byte as a signed number and thus sign-extends to fill the 24 left-most bits of the register,
  - ***load byte unsigned (lbu):*** works with unsigned integers.

Examples: 8-bit to 16-bit

+2: 0000 0010 => 0000 0000 0000 0010

–2: 1111 1110 => 1111 1111 1111 1110

Logical Operators



Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	`sll`
Shift right	>>	>>>	`srl`
Bit-by-bit AND	&	&	`and, andi`
Bit-by-bit OR	|	|	`or, ori`
Bit-by-bit NOT	~	~	`nor`

Shift Operations

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

- 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)

Shift

```
sll  $t2,$s0,4  # reg $t2 = reg $s0 << 4 bits
```

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001_{two} = 9_{ten}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001\ 0000_{two} = 144_{ten}$

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)

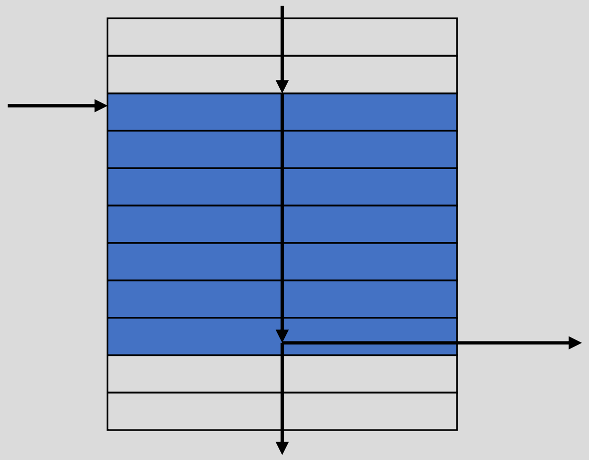
```
nor $t0, $t1, $zero
```

Register 0: always read as zero

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

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

Labels

- In MIPS assembly, a label is simply a string used to name a location in memory.
- A label may refer to the location of a data value (variable) or of an instruction.
- Think of a label as representing an address
- Labels are terminated by a colon character

Complete MIPS Example

```
.data
N:      .word 10

.text

main:
        lw   $t0, N         # $t0 <-- Mem[N]  (10)
        la   $t1, N         # $t1 <-- N       (address)
        . . .
exit:   li   $v0, 10
        syscall
```

Unconditional Branch Instructions

- `j Label          # PC = Label`
- `b Label          # PC = Label`
- `jr $ra           # PC = $ra`
- These help with constructing loops

Making Decisions

- MIPS assembly language includes two decision-making instructions, similar to an if statement with a go to.

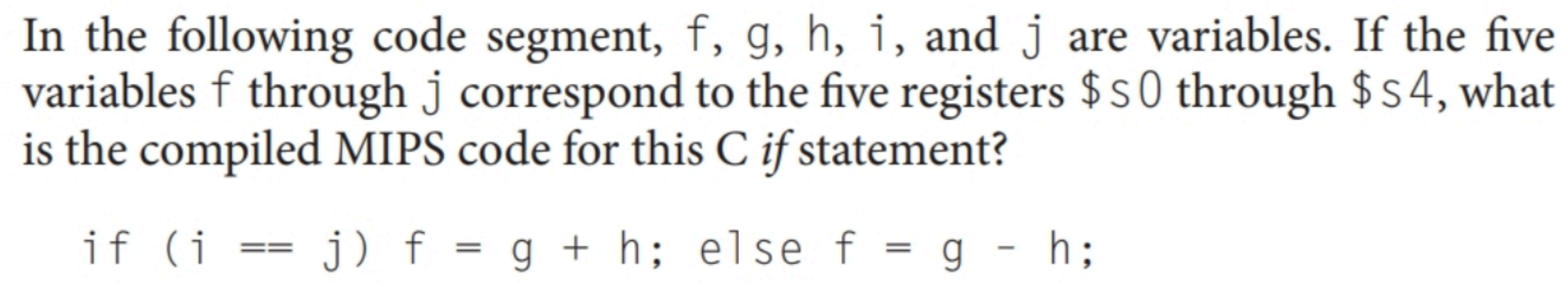
```
beq register1, register2, L1
```

- branch if equal – go to L1 if register1 is equal to register2

```
bne register1, register2, L1
```

- branch if not equal

Example: Decisions


In the following code segment, `f`, `g`, `h`, `i`, and `j` are variables. If the five variables `f` through `j` correspond to the five registers `$s0` through `$s4`, what is the compiled MIPS code for this C *if* statement?

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

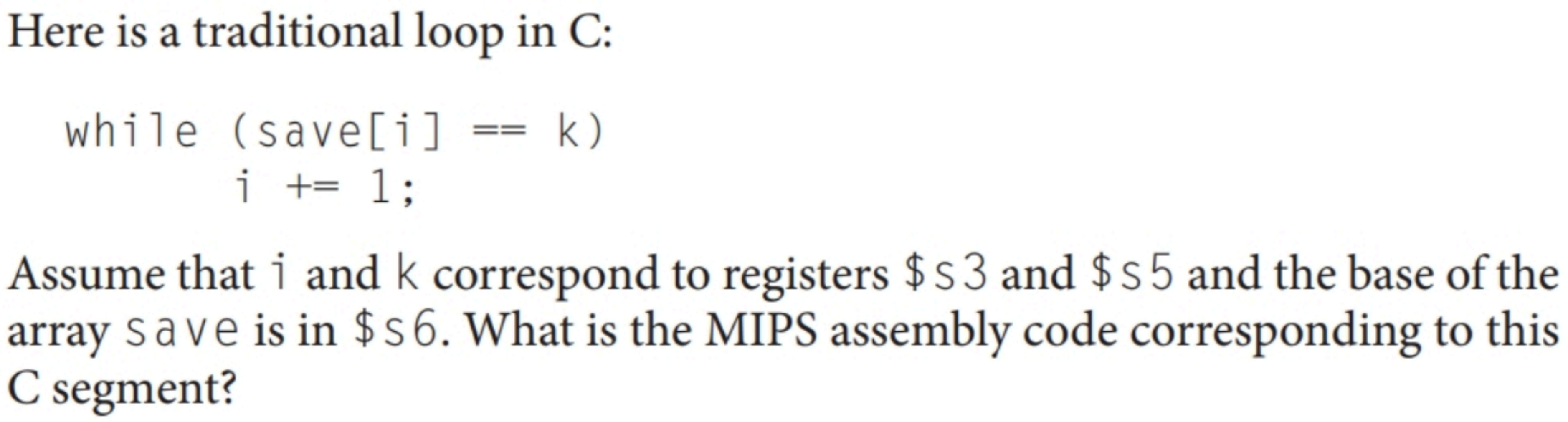
Solution

```
bne $s3,$s4,Else    # go to Else if i ≠j
add $s0,$s1,$s2     # f = g + h (skipped if i ≠j)

j Exit    # go to Exit

Else: sub $s0,$s1,$s2 # f = g – h (skipped if i = j)
Exit:
```

Example: Loops



Here is a traditional loop in C:

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

Assume that `i` and `k` correspond to registers `$s3` and `$s5` and the base of the array `save` is in `$s6`. What is the MIPS assembly code corresponding to this C segment?

Loop

1. load save[i] into a temporary register.
   - Base Address of save
   - Multiply i to get the byte address of the index (by 4)

2. Loop test
   - Choose to use the bne or beq

3. Make branch labels for each portion of the loop test

Solution: loops

```
Loop: sll $t1,$s3,2        # Temp reg $t1 = i * 4
      add $t1,$t1,$s6      # $t1 = address of save[i]
      lw $t0,0($t1)        # Temp reg $t0 = save[i]
      bne $t0,$s5, Exit    # go to Exit if save[i] ≠ k
      addi $s3,$s3,1       # i = i + 1
      j  Loop  # go to Loop

Exit:
```

set on less than

```
slt  $t0, $s3, $s4  # $t0 = 1 if $s3 < $s4
```

```
slti $t0,$s2,10# $t0 = 1 if $s2 < 10
```

Note: useful for for loops

NOTE: MIPS compilers use the slt, slti, beq, bne, and the fixed value of 0 (always available by reading register $zero) to create all relative conditions: equal, not equal, less than, less than or equal, greater than, greater than or eq

Example Switch Statements

- The simplest way to implement switch is via a sequence of conditional tests, turning the switch statement into a chain of if-then-else statements.

```
switch(k){
case 0:  f=i+j;break;
case 1:  f=g+h;break;
case 2:  f=g-h;break;
case 3:  f=i-j;break;
}
```

What is the MIPS assembly code assuming f-k correspond to registers \$s0-\$s5 and \$t2 contains 4 and \$t4 contains base address of JumpTable?

```
slt $t3,$s5,$zero   # test if k=4
add $t1,$s5,$s5     #$t1 =2*k
add $t1,$t1,$t1     #$t1 =4*k
add $t1,$t1,$t4     #$t1=address of JumpTable[k]
lw $t0,0($t1)       #$t0=JumpTable[k]
jr $t0              #jump based on register $t0
L0:  add $s0,$s3,$s4
     j Exit
L1:  add $s0,$s1,$s2
     j Exit
L2:  sub $s0,$s1,$s2
     j Exit
L3:  sub $s0,$s3,$s4

Exit:
```

Procedures in MIPS

function is one tool programmers use to structure programs, both to make them easier to understand and to allow code to be reused.

1. Put parameters in a place where the procedure can access them.
2. Transfer control to the procedure.
3. Acquire the storage resources needed for the procedure.
4. Perform the desired task.
5. Put the result value in a place where the calling program can access it.
6. Return control to the point of origin, since a procedure can be called from several points in a program.

Procedures in MIPS

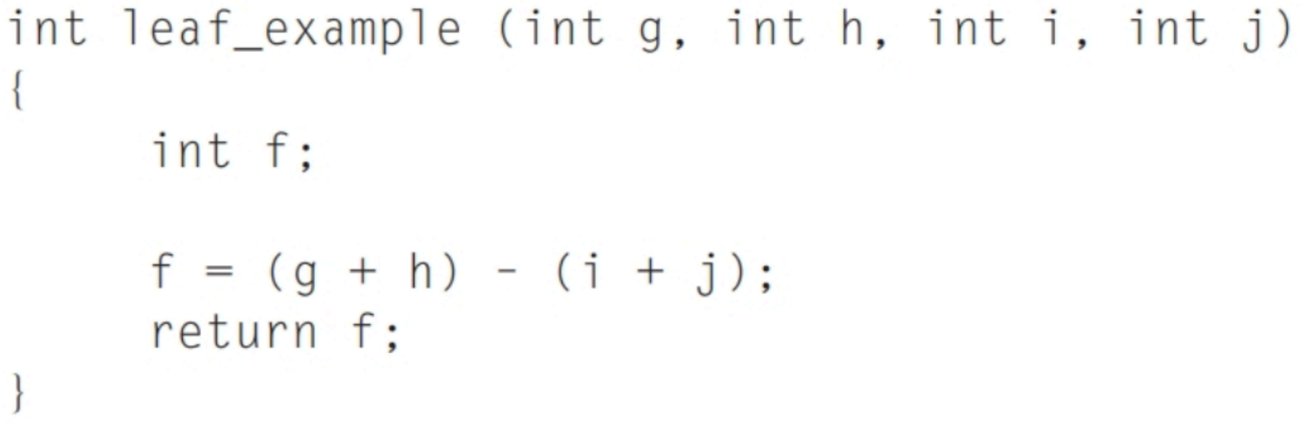
- **$a0–$a3:** four argument registers in which to pass parameters
- **$v0–$v1:** two value registers in which to return values
- **$ra:** one return address register to return to the point of origin
- **jump-and-link instruction (jal):** it jumps to an address and simultaneously saves the address of the following instruction in register `$ra` `(register 31)`

Stack Pointer

- Suppose a compiler needs more registers for a procedure than the four argument and two return value registers.
- Any registers needed by the caller must be restored to the values that they contained before the procedure was invoked.
- The ideal data structure for spilling registers is a *stack*—a last-in-first-out.
- Stack pointer is adjusted by one word for each register that is saved or restored. MIPS software reserves register 29 for the stack pointer, giving it the obvious name $sp.

Procedure Example

The parameter variables g, h, i, and j correspond to the argument registers $a0, $a1, $a2, and $a3, and f corresponds to $s0.



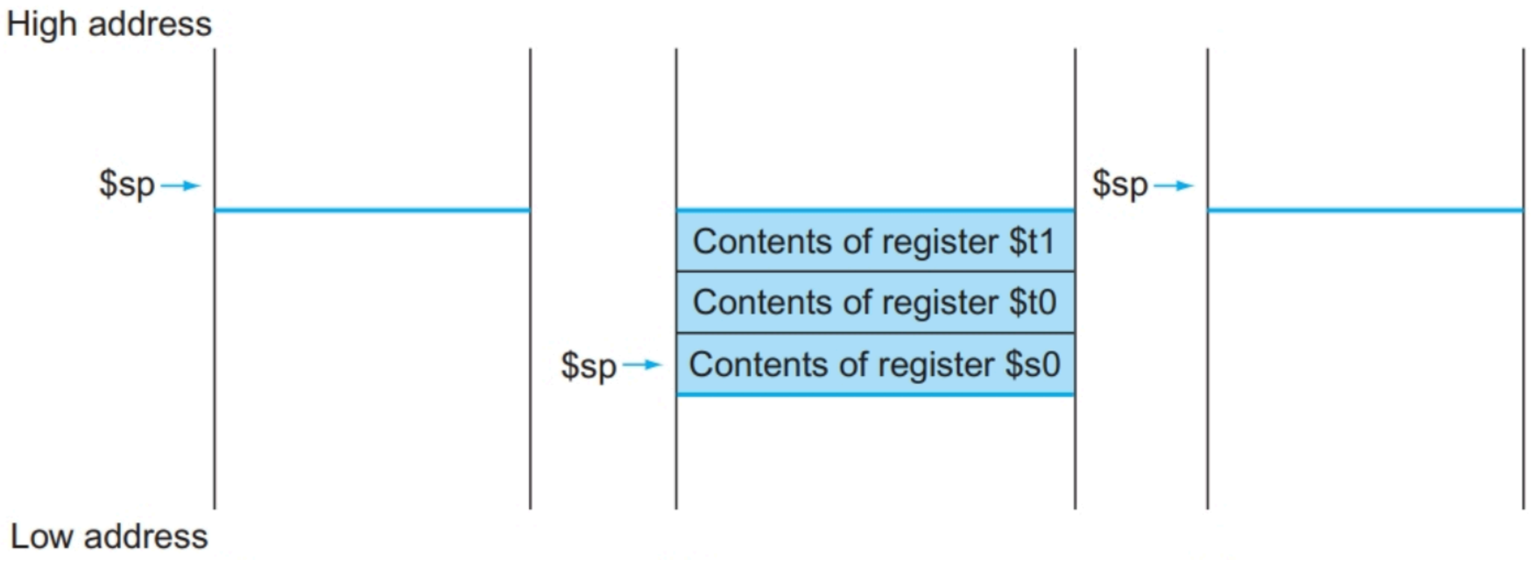
```
int leaf_example (int g, int h, int i, int j)
{
     int f;

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

Leaf Procedure Example

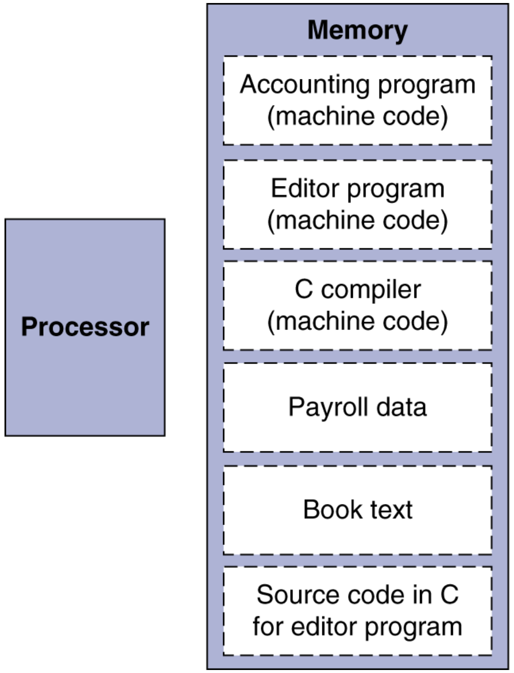
- MIPS code:

```
leaf_example:
  addi $sp, $sp, -4
  sw   $s0, 0($sp)      Save $s0 on stack
  add  $t0, $a0, $a1
  add  $t1, $a2, $a3    Procedure body
  sub  $s0, $t0, $t1
  add  $v0, $s0, $zero  Result
  lw   $s0, 0($sp)
  addi $sp, $sp, 4      Restore $s0
  jr   $ra              Return
```

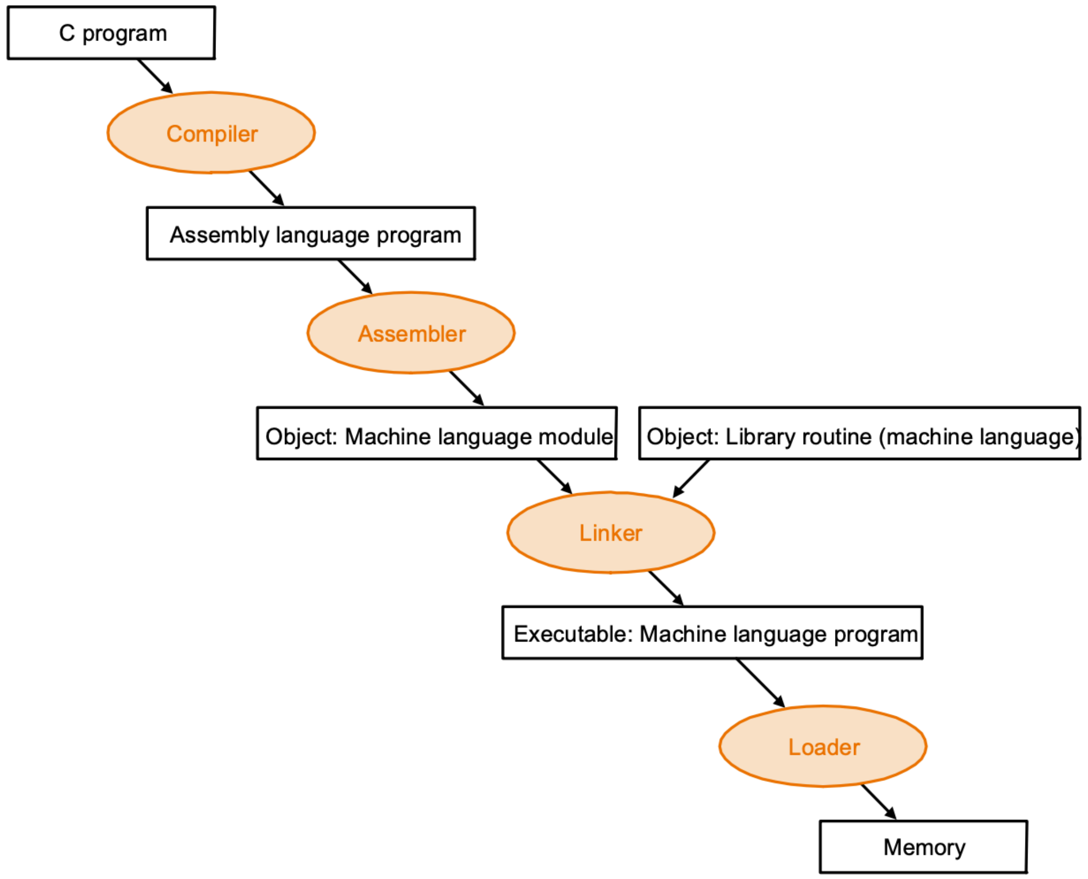

High address
$sp
$sp
Contents of register $t1
Contents of register $t0
Contents of register $s0
$sp
Low address

Stored Program Computers

The BIG Picture


Memory
Accounting program (machine code)
Editor program (machine code)
C compiler (machine code)
Payroll data
Book text
Source code in C for editor program
Processor

- 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


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory