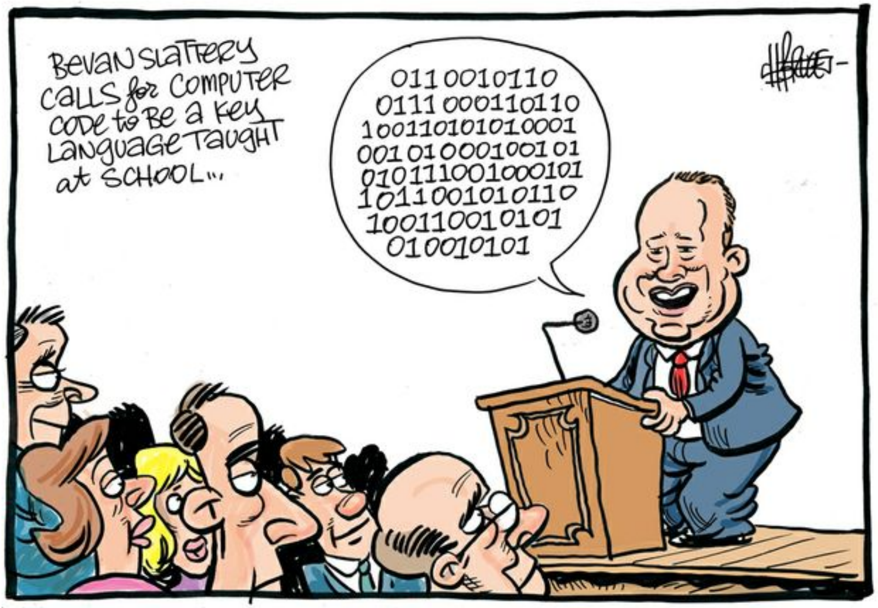
COMPUTER ARCHITECTURE

Chapter 2: Instruction set architecture - ISA

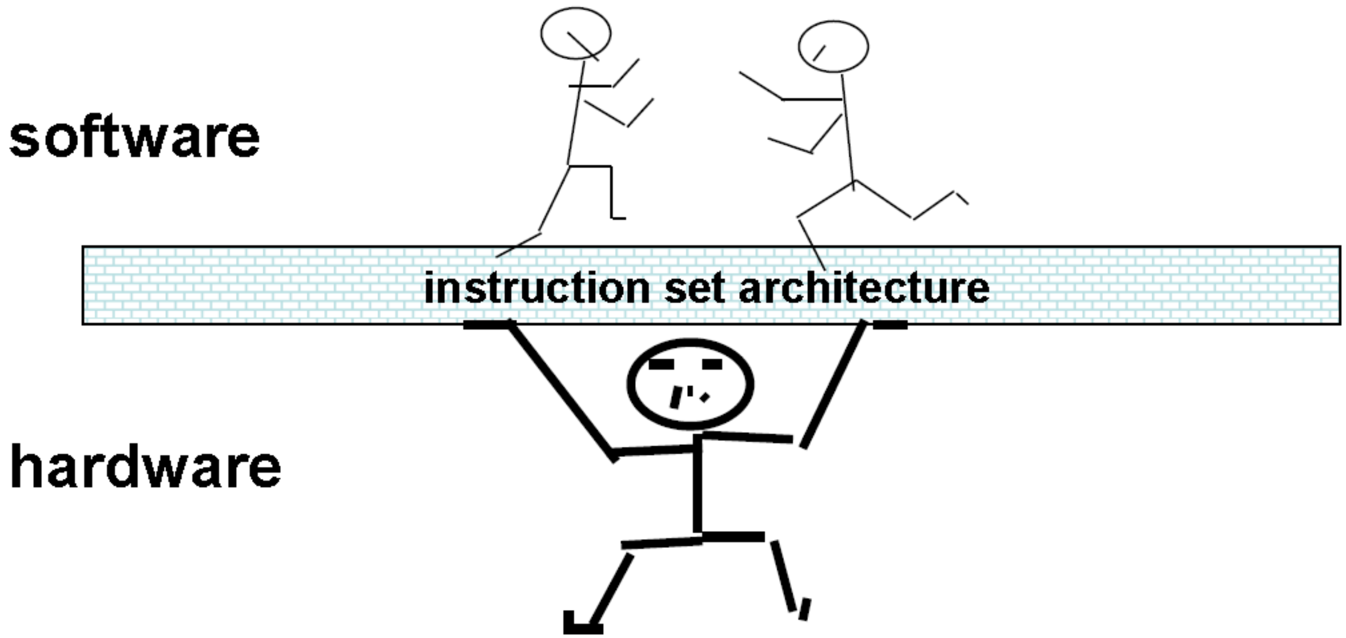
Introduction

- Q: Why do we need to learn the language of computers?
- A: To command a computer's hardware: speak its language



Source: http://media.apnarm.net.au/img/media/images/2013/08/14/computer_language_t620.jpg

Instruction set architecture



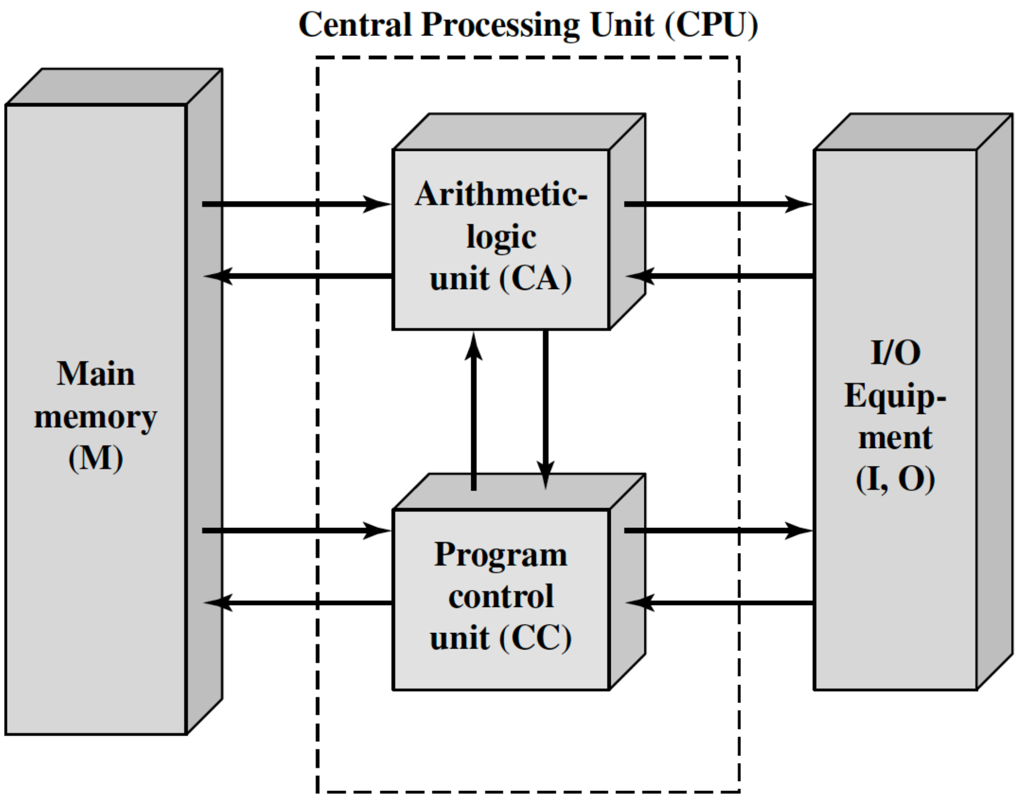
MIPS32 Add Immediate Instruction

001000	00001	00010	0000000101011110
OP Code	Addr 1	Addr 2	Immediate value

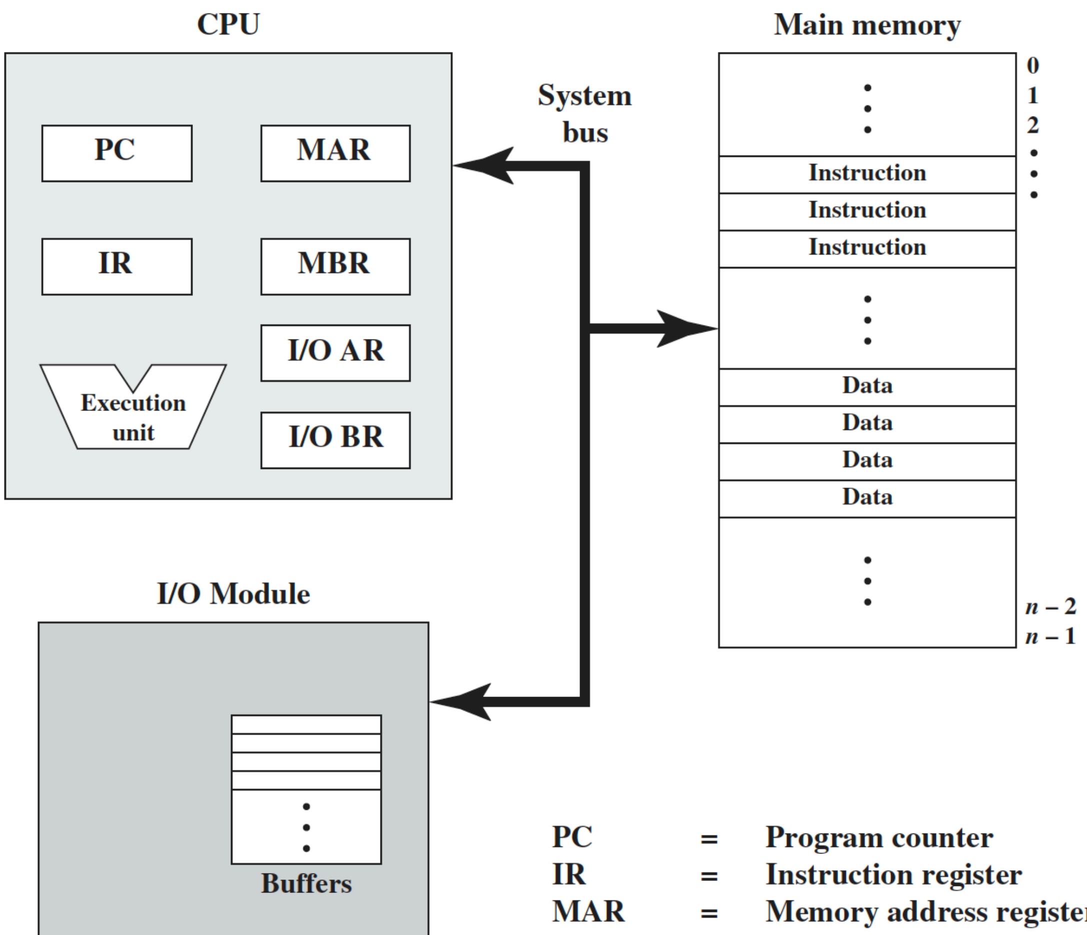
Equivalent mnemonic: **addi $r1 , $r2 , 350**

Von Neumann architecture

- Stored-program concept
- Instruction category:
  - Arithmetic
  - Data transfer
  - Logical
  - Conditional branch
  - Unconditional jump

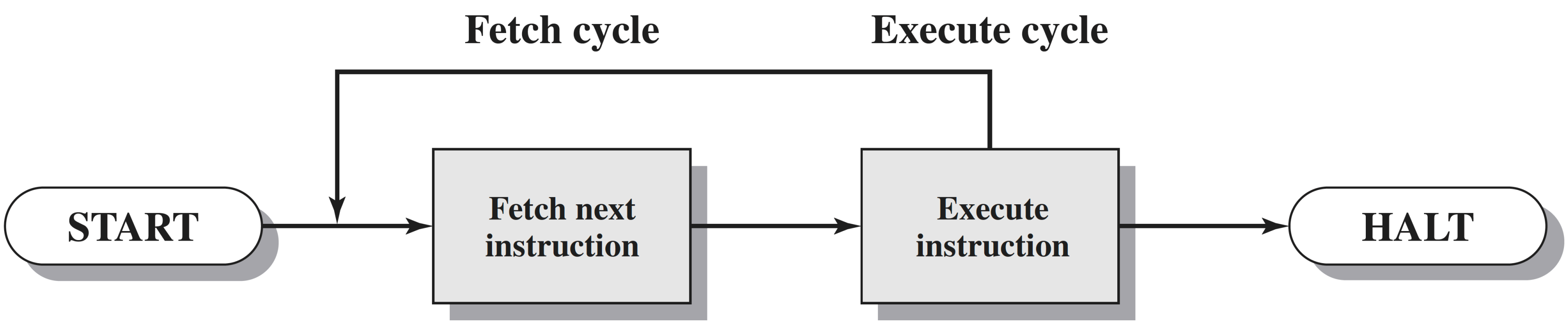


Computer components



PC	=	Program counter
IR	=	Instruction register
MAR	=	Memory address register
MBR	=	Memory buffer register
I/O AR	=	Input/output address register
I/O BR	=	Input/output buffer register

Instruction execution model



- **Instruction fetch**: from the memory
  - PC increased
  - PC stores the next instruction
- **Execution**: decode and execute

STANDARD MIPS INSTRUCTIONS

MIPS instruction set

- MIPS architecture
- MIPS Assembly Instruction ⇔ MIPS Machine Instruction
- Assembly:
  - add $t0, $s2, $t0
- Machine:
  - 000000_10010_01000_01000_00000_100000
- Only one operation is performed per MIPS instruction
  - e.g., $a + b + c$ needs at least two instructions

Instruction set design principle

- Simplicity favors regularity
- Smaller is faster
- Make the common case fast
- Good design demands good compromises

MIPS operands

1. **Register**: 32 32-bit registers (start with the **$** sign)
   - $s0-$s7: corresponding to variables
   - $t0-$t9: storing temporary value
   - $a0-$a3
   - $v0-$v1
   - $gp, $fp, $sp, $ra, $at, $zero, $k0-$k1
2. **Memory operand**: 2^{30} memory words (4 byte each): accessed only by data transfer instructions
3. **Short integer immediate**: -10, 20, 2020,…

Only three operand types! Nothing else!

1st group: arithmetic instructions

- Assembly instruction format:

Opcode	Destination register	Source register 1	Source register 2(*)

- Opcode:
  - add: DR = SR1 + SR2
  - sub: DR = SR1 – SR2
  - addi: SR2 is an immediate (e.g. 20), DR = SR1 + SR2
- Three register operands

Example

- **Question**: what is MIPS code for the following C code

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

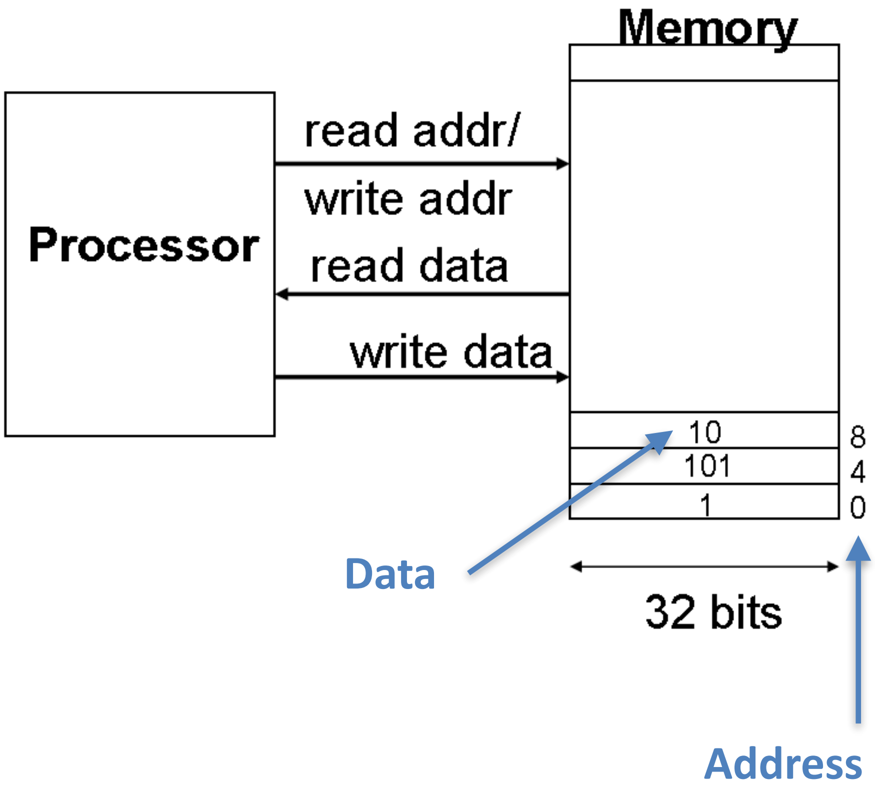
- If the variables g, h, i, j, and f are assigned to the register $s0, $s1, $s2, $s3, and $s4, respectively.
- **Answer**:

```
add $t0, $s0, $s1 # g + h
add $t1, $s2, $s3 # i + j
sub $s4, $t0, $t1 # t0 - t1
```

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

2^{nd} group: data transfer instructions

- **Copy data** b/w memory and registers in CPU
  - Register
  - Address: a value used to delineate the location of a specific data element within a memory array
- Load (l): **copy data** from memory to a register
- Store (s): **copy data** from a register to memory



Data transfer instructions

- Assembly instruction format:

Opcode	Register	Memory operand

- Opcode:
  - Size of data: 1 byte, 2 bytes, or 4 bytes (word)
  - Behaviors: load or store
- Register:
  - Load: destination
  - Store: source
- Memory operand: `offset(base register)`
  - `offset`: short integer number
  - Byte address: each address identifies an 8-bit byte
  - “words” are aligned in memory (address must be multiple of 4)

Memory operand

$$\text{Memory address} = < \text{offset} > + \text{value}(< \text{base register} >)$$

- **Question**: given the following memory map, assume that $\$s0$ stores value of 8. Which is the memory operand used to access the byte storing value of 0x9A?

address:	8	9	10	11
	0x12	0x34	0x56	0x78
	0x9A	0xBC	0xDE	0xF0
address:	12	13	14	15

- **Answer**:
  - Address of the byte storing value of 0x9A is 12
  - If we use $\$s0$ as the base register, offset $= 12 - 8 = 4$
  - Memory operand: $4(\$s0)$

Load instructions

- **Remind**: "load" means copying data from memory to a 32-bit register
- Instructions:
  - lw: load word
    - eg.: lw $s0, 100($s1) #copy 4 bytes from memory to $s0
  - lh: load half - sign extended
    - eg.: lh $s0, 100($s1) #copy 2 bytes to $s0 and extend signed bit
  - lhu: load half unsigned - zero extended
  - lb: load byte - sign extended
  - lbu: load byte unsigned - zero extended
  - Special case: lui - load upper immediate
    - eg.: lui $s0, 0x1234 #$s0 = 0x12340000

Store instructions

- **Remind**: "store" means copying data from a register to memory
- Instructions:
  - sw: store word
    - eg.: sw $s0, 100($s1) #copy 4 bytes in $s0 to memory
  - sh: store half - two least significant bytes
    - eg.: sh $s0, 100($s1) #copy 2 bytes in $s0 to memory
  - sb: store byte - the least significant byte
    - eg.: sb $s0, 100($s1) #copy 1 bytes in $s0 to memory

Memory operands

- Main memory used for composite data
  – Arrays, structures, dynamic data
- To apply arithmetic operations
  – Load value(s) from memory into register(s)
  – Apply arithmetic operations to the register(s)
  – Store result from a register to memory (if required)
- MIPS is Big Endian
  – Most-significant byte at least address of a word
  – Little Endian: least-significant byte at least address

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
```

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
```

Exercises

1. Given the following memory map, assume that the register $t0 stores value 8 while $s0 contains 0xCAFEFACE. Show the effects on memory and registers of following instructions:

    a) lw $t1, 0($t0)
    b) lw $t2, 4($t0)
    c) lh $t6, 4($t0)
    d) lb $t5, 3($t0)
    e) sw $s0, 0($t0)
    f) sb $s0, 4($t0)
    g) lh $s0, 7($t0)

address:	8	9	10	11
	0x12	0x34	0x56	0x78
	0x9A	0xBC	0xDE	0xF0
address:	12	13	14	15

Exercises

2. Convert the following C statements to equivalent MIPS assembly language if the variables f, g, and h are assigned to registers $s0, $s1, and $s2 respectively. Assume that the base address of the array A and B are in registers $s6 and $s7, respectively.
   a) f = g + h + B[4]
   b) f = g - A[B[4]]

3rd group: logic instructions

- Instruction format: the same with arithmetic instructions
- Bitwise manipulation
  - Process operands bit by bit

Operation	C operator	MIPS opcode
Shift left	<<	sll
Shift right	>>	srl
Bitwise AND	&	and, andi
Bitwise OR	|	or, ori
Bitwise NOT	~	nor

Shift operations

- Shift left (sll)
  - Shift value in the first source to left and fill least significant positions with 0 bits
  - e.g., sll $s0, $s1, 4 # $s0 = $s1 << 4
  - Special case: sll by i bits multiplies by 2^i
- Shift right (srl)
  - Shift value in the first source to right and fill most significant positions with 0 bits
  - e.g., srl $s0, $s1, 4 # $s0 = $s1 >> 4
  - Special case: srl by i bits divides by 2^i (unsigned number only)

AND operation

- Bitwise AND two source operands
  - and: two registers
    - e.g., and $s0, $s1, $s2 #$s0 = $s1 & $s2
  - andi: the second source is a short integer number (16 bit)
    - 16 high-significant bits of the result are 0s
    - e.g., andi $s0, $s1, 100 #$s0 = {16'b0,$s0[15:0]&100}
- Useful to mask bits in a register
  - Select some bits and clear others to 0

OR operation

- Bitwise OR two source operands
  - or: two registers
    - e.g., or $s0, $s1, $s2 # $s0 = $s1 | $s2
  - ori: the second source is a short integer number (16 bit)
    - Copy 16 high-significant bit from the first source to destination
    - e.g., ori $s0, $s1, 100 # $s0 = {$s1[31:16],$s1[15:0]|100}
- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

NOT operation

- Don't have a **not** instruction in MIPS ISA
  - 2-operand instruction
  - Useful to invert all bits in a register
- Can be done by the **nor** operator, 3-operand instruction
  - a NOR b = NOT (a OR b)
  - NOT a = NOT (a OR 0) = a NOR 0
  - e.g., `nor $s0, $s0, $zero`
  - What else?

Example

- **Question**: assume that \$s0 and \$s1 are storing values 0x1234567 and 0xCAFEFACE, respectively. What is value of \$S2 after each following instructions
  1. sll \$s2, \$s0, 4
  2. and \$s2, \$s0, \$s1
  3. or \$s2, \$s0, \$s1
  4. andi \$s2, \$s0, 2020
- **Answer**:
  1. 0x23456780
  2. 0x02345248
  3. 0xDAFEFEFE
  4. 0x00000660

Exercise

- Find the value for $t2 after each following sequence of instructions if the values for register $t0 and $t1 are
  a) $t0 = 0xAAAAAAAA, $t1 = 0x12345678
  b) $t0 = 0xF00DD00D, $t1 = 0x11111111

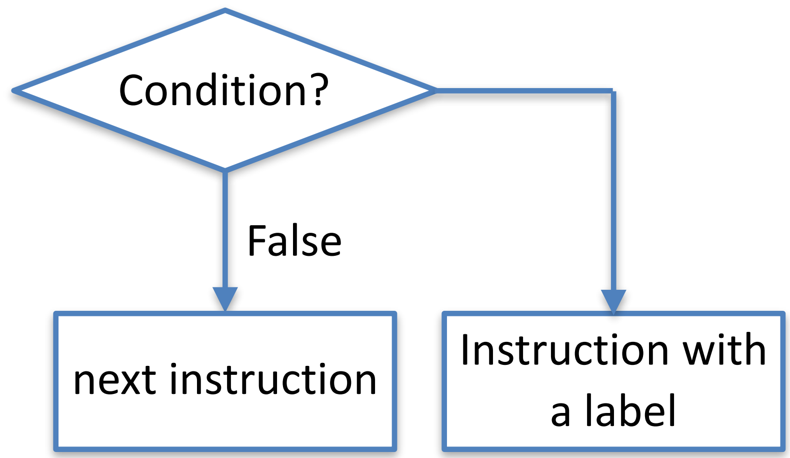
Sequence 1:
```
sll $t2, $t0, 44
or $t2, $t2, $t1
```

Sequence 2:
```
sll $t2, $t0, 4
andi $t2, $t2, -2
```

Sequence 3:
```
srl $t2, $t0, 3
andi $t2, $t2, 0xFFEF
```

4th group: conditional branch instructions

- Branch to a **label** if a condition is true; otherwise, continue sequentially
- Only two standard conditional branch instructions
  - beq $rs, $rt, L1 #branch if equal
    - If (rs == rt), go to L1
  - bne, $rs, $rt, L1
    - If (rs != rt, go to L1
- Label: a given name
  - format: <label>: <instruction>




```
    addi $t0, $zero, 0
    addi $t1, $zero, 5
L1: addi $t0, $t0, 1
    bne $t0, $t1, L1    #branch to L1
    add $t0, $t1, $t0
```

5^{th} group: unconditional jump instructions

- Immediately jump to a label
  - Without any condition checked
- Three standard unconditional jumps
  - j L1
    - Jump to the label L1
  - jal L1
    - Jump to the label L1 and store address of the next instruction to the $ra register
    - Used for function/procedure call
  - jr register
    - Jump to an instruction whose address is stored in the register
    - Used for returning to the caller function/procedure from a sub-function/-procedure

Example

- **Question**: Compile the following C code into MIPS code (assume that f, g, h, i, and j are stored in registers from $s0 to $s4, respectively)

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

- **Answer**:

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

```
      beq $s3, $s4, If
      sub $s0, $s1, $s2
      j Exit
If:   add $s0, $s1, $s2
Exit:
```

- How can we compare less than or greater than?
  - e.g., if (a < b)

Exercise

- Convert the following C code to MIPS. Assume that the base address of the save array is stored in $s0 while i and k are stored in the registers $s1 and $s2, respectively

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

Set-on-less-than instruction

- Used for comparing less than or greater than
  - Results (destination registers) are always 0 (false) or 1 (true)
- slt $rd, $rs, $rt
  - if (rs < rt) rd = 1; else rd = 0;
- slti $rt, $rs, immediate
  - if (rs < immediate) rt = 1; else rt = 0;
- sltu $rd, $rs, $rt
  - Values are unsigned number
- sltui $rt, $rs, immediate
  - Values in registers are unsigned number

Example-1

- **Question**: Assume that values storing in $s1 and $s2 are 0xFFFFFFFF and 0x00000001, respectively. What are the results in $t0 and $t1 after the following instructions

```
slt $t0, $s1, $s2
sltu $t1, $s1, $s2
```

- **Answer**:
  - $t0 = 1 due to signed numbers comparison ($s1 = -1)
  - $t1 = 0 due to unsigned numbers comparison ($s1 = 4.294.967.295)

Example-2

- **Question**: Compile the following C code into MIPS code (assume that f, g, h, i, and j are stored in registers from $s0 to $s4, respectively)

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

- **Answer**:

```
      slt $t0, $s3, $s4
      beq $t0, $zero, Else
      add $s0, $s1, $s2
      j Exit
Else: sub $s0, $s1, $s2
Exit:
```

```
      slt $t0, $s3, $s4
      bne $t0, $zero, If
      sub $s0, $s1, $s2
      j Exit
If:   add $s0, $s1, $s2
Exit:
```

-

Exercise

- Convert the following C code to MIPS instructions
  a) Sequence 1:

```
int a, b;
…
if (a >= 0) a = a + b;
else a = a - b;
```

  b) Sequence 2:

```
int a, i;
for (i = 0, a = 0; i < 10; i++)
    a++;
```

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

Pseudo instructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudo instructions (instructions in blue): figments of the assembler's imagination
  - Help programmer
  - Need to be converted into standard instructions
- For example
  - move $t0, $t1 → add $t0, $zero, $t1
  - blt $t0, $t1, L → slt $at, $t0, $t1
    bne $at, $zero, L

PROCEDURE CALL

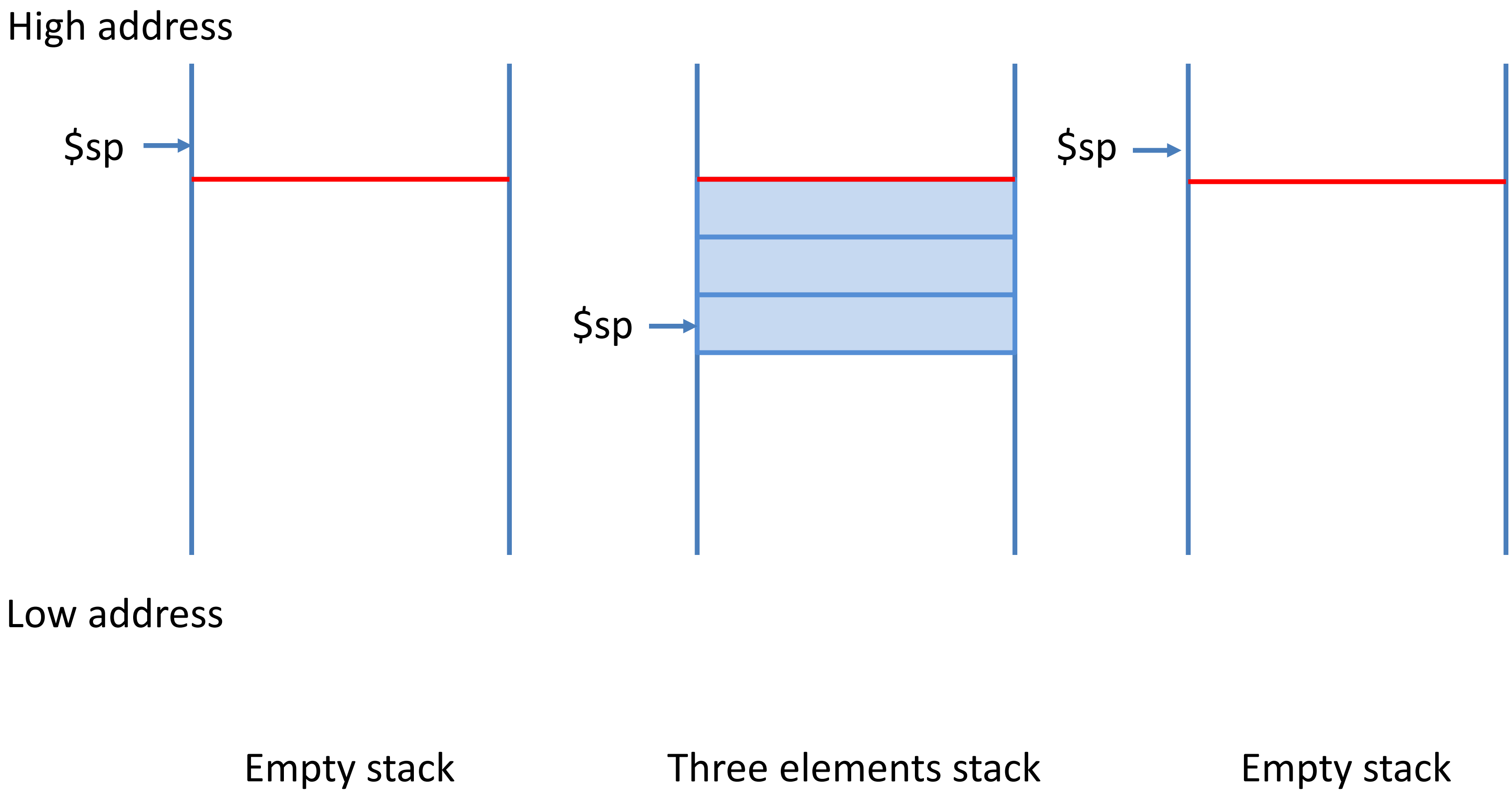
Procedure calling

- Caller vs. Callee
- Steps required to call a procedure
  - Place parameters in registers
  - Transfer control to procedure
  - Acquire storage for procedure
  - Perform procedure’s operations
  - Place result in register for caller
  - Return to place of call

Register usage

- $a0 – $a3: arguments
- $v0, $v1: result values
- $t0 – $t9: temporaries
  - Can be overwritten by callee
- $s0 – $s7: saved
  - Must be saved/restored by callee
-
- $gp: global pointer for static data
- $sp: stack pointer
- $fp: frame pointer
- $ra: return address

Stack addressing model



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

- **Leaf-procedure**: will not call any sub-procedure or itself
  - No need to care any else, except saved registers
- 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 - MIPS code

leaf_example:	**Label procedure name (used to call)**
addi \$sp, \$sp, -4 sw \$s0, 0(\$sp)	**Save \$s0 to the stack**

add \$t0, \$a0, \$a1
add \$t1, \$a2, \$a3
sub \$s0, \$t0, \$t1
	Procedure body

add \$v0, \$s0, \$zero	**Result**
lw \$s0, 0(\$sp) addi \$sp, \$sp, 4	**Restore \$s0**
jr \$ra	**Return to caller**

Non-leaf procedure

- Non-leaf procedure: will call at least an other procedure or itself
  - Need to care:
    - Return address ($ra)
      - Caller call A ($ra = caller)
      - A call B ($ra = A)
      - B return to A ($ra = A)
      - Cannot return to caller
    - Arguments transferred from caller if needed
  - Use the stack to backup

Non-leaf procedure - example

- Factorial calculation
- C code

```c
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 - 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
```

No recursive call

- **No change in $a0 & $ra**
- **No need to restore from stack**

Recursive call

Exercise

- Write corresponding MIPS code for the following function

```
void strcpy (char x[], char y[]){
  int i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

  – Addresses of x, y in $a0, $a1
  – i in $s0
  – Ascii code of '\0' is 0

MACHINE INSTRUCTIONS

Machine 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!
- Representing instructions:
  - Instruction format: R, I, and J
  - Opcode: predefined (check the reference card)
  - Operands:
    - Register numbers: predefined (check the reference card)
    - Immediate: integer to binary
    - Memory operands: offset + base register

Instruction formats

- MIPS instructions use one of three formats
  - **R-format**: encoding all-register-operands instructions and two shift instructions
    - add $s0, $s1, $s2 # all operand are registers
    - sll $s0, $s1, 4 # shift instruction
  - **I-format**: encoding instructions with one operand different from registers
    - **except**: sll, srl, j, and jal
    - addi $s0, $s1, 100 #immediate
    - lw $s1, 100($s0) #memory operand
  - **J-format**: encoding j and jal
    - j Label1

R-format instructions

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

- op: always 0 for R-format instructions
  – 6 bits for all formats
- rs: first source register number
- rt: second source register number
- rd: destination register number
- shamt: shift amount (0 for non-shift instructions)
- funct: identify operators

Register numbers & some function fields

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

Instruction	Function field	Instruction	Function field
add	0x20 (32)	sub	0x22 (34)
and	0x24 (36)	or	0x25 (37)
nor	0x28 (39)	jr	0x08 (8)
sll	0x00 (0)	srl	0x02 (2)
slt	0x2A(42)	sltu	0x2B (43)

Example-1

- **Question**: what is machine code of following instruction

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

rd rs rt

- **Answer**:
  - R-format is used to encode the above instruction

0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

  - Machine code: 0x02324020

Example-2

- **Question**: what is the assembly instruction of the following MIPS machine code

0000_0000_0001_0000_0101_0001_0000_0000

- **Answer**:
  - Opcode (the 6 high-significant bits) is 0 ⇒ an R-format instruction

000000	00000	10000	01010	00100	000000

  - Function field is 0 ⇒ a sll instruction
  - The assembly instruction: sll $t2, $s0, 4
- **Note**: shift instructions don't use the rs field

MIPS I-format Instructions

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

- **op**: specific values for instructions
- **rs**: source or base address register (**no destination**)
  - First source register in 2 source register instructions
- **rt**: source or destination register
  - Second source register in 2 source register instructions
- **constant/address**: $-2^{15} \rightarrow 2^{15} - 1$

Some opcode fields

Instruction	Opcode field	Instruction	Opcode field
addi	0x08 (8)	addiu	0x09 (9)
lbu	0x24 (36)	lhu	0x25 (37)
lb	0x20 (32)	lh	0x21 (33)
lw	0x23 (35)	sw	0x2B (43)
sb	0x28 (40)	sh	0x29 (41)
slti	0x0A (10)	sltiu	0x0B (11)
andi	0x0C (12)	ori	0x0D (13)
beq	0x04 (4)	Bne	0x05 (5)

Example-1

- **Question**: what is machine code of following instruction

lw $t0, 32($s3)

rt (destination) rs (base register)

- **Answer**:
  - I-format is used to encode the above instruction

35	19	8	32

100011	10011	01000	0000_0000_0010_0000

  - Machine code: 0x8E680020

Example-2

- **Question**: what is the assembly instruction of the following MIPS machine code

1010_1101_0010_1000_0000_0110_0000_0100

- Answer:
  - Opcode $\neq$ 0 (not j and jal) $\Rightarrow$ an I-format

101011	01001	01000	0000_0000_0110_0100

  - Opcode = 101011 $\Rightarrow$ sw instruction
  - The assembly instruction: sw $t0, 100($t1)

Exercise

- Write MIPS code for the following C code, then translate the MIPS code to machine code. Assume that $t1 stores the base of array *A* (array of integers) and $s2 stores h

```
int A[301];
int h;
…
A[300] = h + A[300] - 2;
```

Branch & jump addressing

- **Question**: how can we represent labels in conditional branch and unconditional jump instructions?
- **Answer**: use addressing methods
  - Cannot representing label names
  - Calculate distance between the current instruction to the label
    - PC-related addressing
    - (Pseudo) direct addressing

PC-relative addressing

- Use for encoding bne and beq instructions
- Target address (the instruction associated with the label) calculated based on PC - program counter register
  - PC is already increased by 4

$$\boxed{\text{target address} = \text{PC} + \text{address field} \times 4}$$

- When encoding branch conditional instructions, address field should be calculated by

$$\boxed{\text{address field} = \frac{\text{target address} - \text{PC}}{4}}$$

  - The number of instructions from PC to the label

Example-1

- **Question**: given the following MIPS code, what is the machine code for the conditional branch instruction?

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

- **Answer**:
  - I-format is used
  - Assume that the bne instruction is stored at address x
    - PC $= x + 4$ when processing the bne instruction
    - target address $= x + 12 \Rightarrow$ address field $= \dfrac{(x+12)-(x+4)}{4} = 2$

5	19	20	2

    - Machine code: 0x16740002

Example-2

- **Question**: given the following MIPS code, what is the machine code for the conditional branch instruction?

```
Label: addi $t0, $t0, -1
       bne $t0, $t1, Label
       add $t0, $t0, $s1
```

- **Answer**:
  – I-format is used
  – Assume that the bne instruction is stored at address x
    - PC $= x + 4$ when processing the bne instruction
    - target address $= x - 4 \Rightarrow$ address field $= \dfrac{(x-4)-(x+4)}{4} = -2$

5	8	9	-2

16-bit 2's complement

- Machine code: 0x1611FFFE

J-format instructions

- PC-relative and I-format not good enough
  - 16 bit address field allows to jump only $\pm 2^{15}$ instructions

op	address
6 bits	26 bits

- J-format is used for j and jal instructions only
  - opcode: 2 for j; 3 for jal
  - address: used for calculate target address of jump instructions

$$\boxed{\text{target address} = \{\text{PC}[31:28], \text{address}[25:0], 00_2\}}$$

  - Need full address of instructions

Example

- **Question**: given the following MIPS code, what is the machine code for the unconditional jump instruction if the first instruction is stored at memory location 80000?

```
Label: addi $t0, $t0, -1
       bne $t0, $t1, Exit
       j Label
Exit:
```

- **Answer**:
  - J-format is used
    - PC $= 80012$ when processing the j instruction $\Rightarrow$ PC$[31:28] = 0000_2$
    - target address $= 80000 = \{0000_2,$ address field,$00_2\}$
    - $\Rightarrow$ address field $= \frac{80000}{4} = 20000$

2	20000

  - Machine code: 0x08004E20

Exercise

- Decide the machine code of the following sequence. Assume that the first instruction (start label) is stored at memory address 0xFC00000C

Text

```
start:....
loop: addi $t0, $t0, -1
      sw $t0, 4($t2)
      bne $t0, $t3, loop
      j start
```

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:  …
```

Summary

- MIPS ISA
  - 3 types of operands
  - 5 groups of instructions
- Procedure call
- Machine code
  - 3 formats
  - Addressing methods

The end