Instructions: Language of Computer

Outline

- MIPS Assembly Instructions
  - Arithmetic Instruction
  - Data Transfer (Memory Access) Instruction
  - Logical Instruction
  - Conditional Branch
  - Unconditional Jump
- Instruction Encoding

From a High-Level Language to the Language of Hardware

- High-level language
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- Assembly language
  - Textual representation of Machine code
  - strong correspondence between the assemble instructions and the architecture's machine code
- Hardware representation (Machine code)

  Binary digits (bits)

  Encoded instructions and data

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;
```

Compiler

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
```

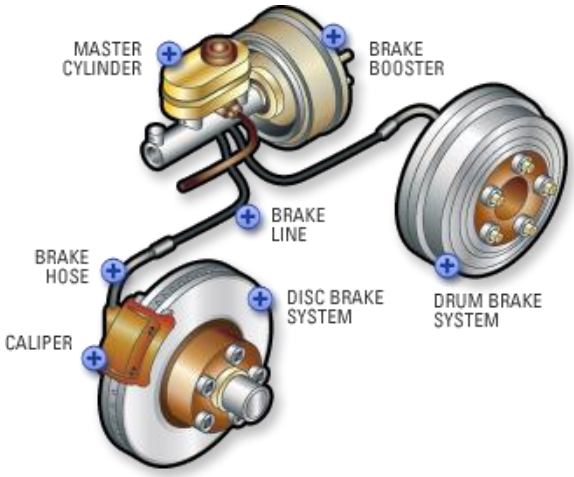
Assembler

Binary machine language program (for MIPS)

```
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

Why do we need to learn Assembly Language?

- Assembly language are more precise than high-level language
  - Close mapping between machine codes and assembly
- Assembly can reveal programs details
  - E.g. Understand how registers and memory are used
- Assembly is very useful when the speed or size of a program is critically important
  - Assembly is able to directly control the hardware
  - Require less memory space and provide predictable timing
  - For example, a computer control a car’ s brakes to respond rapidly and predictably to events in the outside world
  - Easy to understand and learn modern Assembly Language



MIPS Assembly Language is explained in this tutorial

Category	Instruction	Example	Meaning	Format
Arithmetic	add	add $s1, $s2, $s3	$s1 =$s2 +$ s3	R
	sub	sub $s1, $s2, S3	$s1=$s2 -$s3	R
	add immediate	addi $s1, $s2, 20	$s1 = $s2 +20	I
Data Transfer	load word	lw $s1, 20(s2)	$s1=Memory [$2+20]	I
	store word	sw $s1, 20($s2)	Memory [$2+20]=$s1	I
	Load upper immed.	lui $s1, 20	$s1= 20 * 2^{16}	I
Logical	and	and $s1, $s2, $s3	$s1 =$s2 & $s3	R
	or	or $s1, $s2, $s3	$s1=$s2 | $s3	R
	nor	nor $s1, $s2, $s3	$s1= ~($s2 | $s3)	R
Conditional Branch	branch on equal	beq $s1, $s2, 25	If ($s1==$s2) go to PC+4+100	I
	branch on not equal	bne $s1, $s2, 25	If ($s1!=$s2) go to PC+4+100	I
	set on less than	slt $s1, $s2, $s3	If ($s1<s$2) go to PC+4+100	R
Unconditional Jump	Jump	j 2500	goto 10000	J

Types of Assembly Instructions

- Arithmetic Instruction
  - Arithmetic operation, such as
    - Addition: ADD, ADDI
    - Subtraction: SUB
- Data Transfer Instruction
  - Copy data between register and memory, such as
    - Load Word: LW
    - Store Word: SW
- Logical Instruction
  - Logical operation, such as
    - AND: And operation
    - OR: Or operation
- Conditional Branch Instruction
  - Conditionally change the flow of program execution when , such as
    - Branch on Equal: BEQ
    - Branch on Not Equal: BNE
    - Set on Less Than: SLT
- Unconditional Jump Instruction
  - Unconditionally change the flow of program execution
    - Jump: J

Recap: Hexadecimal Representation

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

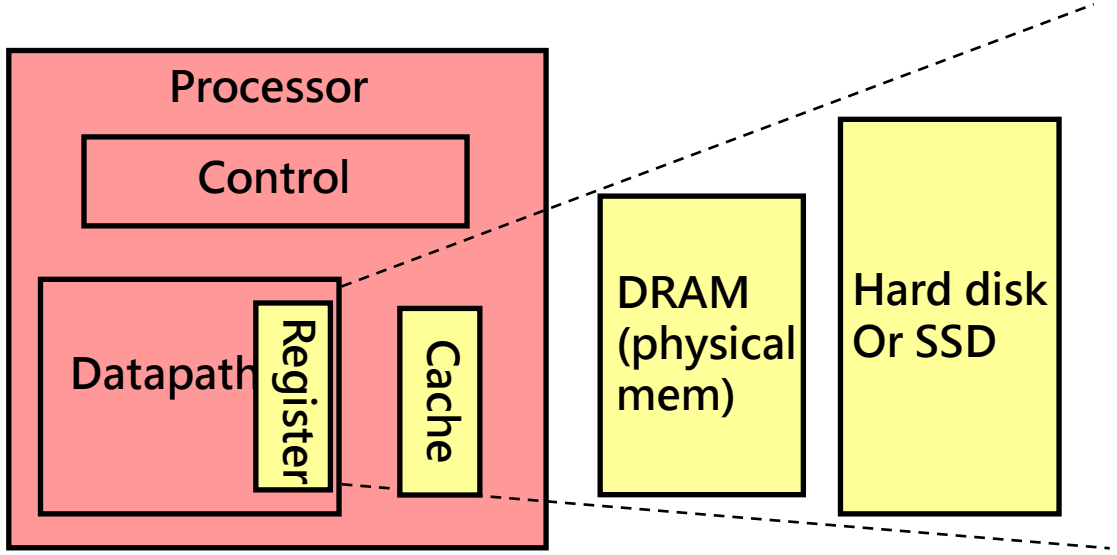
0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

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

Memory Hierarchy- Register

- Memory closer to the processer has faster access speed with smaller sizes
  - Registers are the fastest
- A basic MIPS CPU 32 registers ($0 ~ $31) , and each is 32-bit
  - Names are given to registers based on their functions

e.g.
$s1=$17,
$s2=$18,
$t0=$8

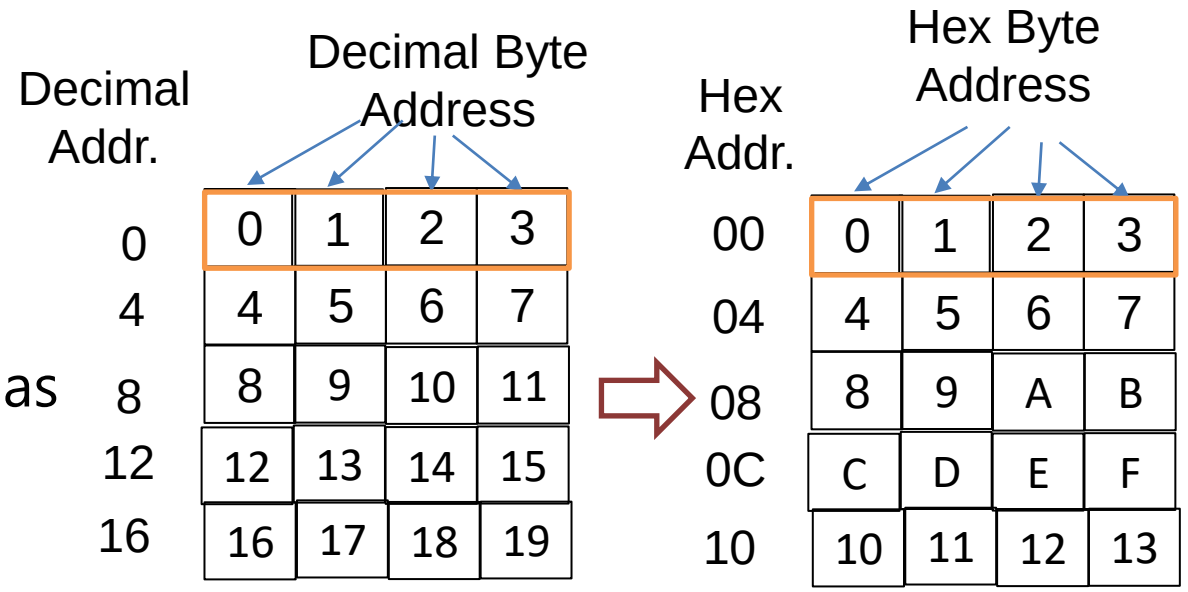


Speed: Fastest → Slowest
Size: Smallest → Biggest
Cost: Highest → Lowest

Name	Register number	Usage
$zero	0	The constant value 0
$v0–$v1	2–3	Values for results and expression evaluation
$a0–$a3	4–7	Arguments
$t0–$t7	8–15	Temporaries
$s0–$s7	16–23	Saved
$t8–$t9	24–25	More temporaries
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address

Memory Hierarchy--Memory

- Main memory (aka DRAM) is used for composite data
  - Arrays, structures, dynamic data
- Memory is byte addressable
  - Each address identifies an 8-bit byte
- Memory address are aligned
  - Normally access 4 bytes at a time (A word has 4 bytes)
  - Word address = Byte address mod 4



e.g., how data 0x12FE34DC are stored in memory?

	MSB			LSB
	12	FE	34	DC
byte	0	1	2	3

e.g., What is the word address of byte 4, 8, and 10?

4 Mode 4 = 1….0
8 Mode 4 = 2
10 mode 4 = 2…2

Arithmetic Instruction -- add & sub

- Both **add** and **sub** instructions performs operations the first and second registers and stores the result in the destination register.
  - Operands are 32-bit registers
- **add** instruction

```
add $rd, $rs, $rt➔$rd = $rs + $rt
```

```
e.g., add $t0, $t2, $t1
      # reg $t0 = $t2 + $t1
```

- **sub** instruction

```
sub $rd, $rs, $rt➔$rd = $rs - $rt
```

```
e.g., sub $t0, $t1, $t2
      #$t0 = $t1 - $t2
```

Registers

$t2	0000 0000 0000 0000 0000 0000 0000 1011
$t1	0000 0000 0000 0000 0000 0000 0000 0010
$t0	0000 0000 0000 0000 0000 0000 0000 1101

$t2	0000 0000 0000 0000 0000 0000 0000 0010
$t1	0000 0000 0000 0000 0000 0000 0000 1011
$t0	0000 0000 0000 0000 0000 0000 0000 1001

Arithmetic Instruction -- addi

- addi performs an addition on both the source register and the immediate data, and stores the result in the destination register

```
addi $rd, $rs, imm➔$rd = $rs +imm
```

```
e.g., addi $t0, $t2, 4
    # reg $t0 = $t2 + 4
```

Registers

$t2	0000 0000 0000 0000 0000 0000 0000 1001

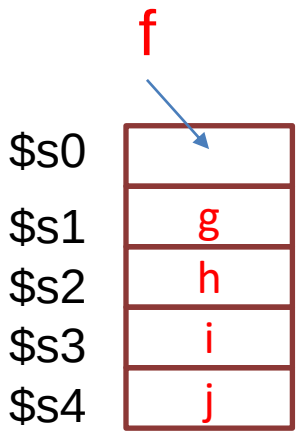
+4 ⇓

$t0	0000 0000 0000 0000 0000 0000 0000 1101

Arithmetic Instruction Example

- Find the compiled MIPS code for the following C code:

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

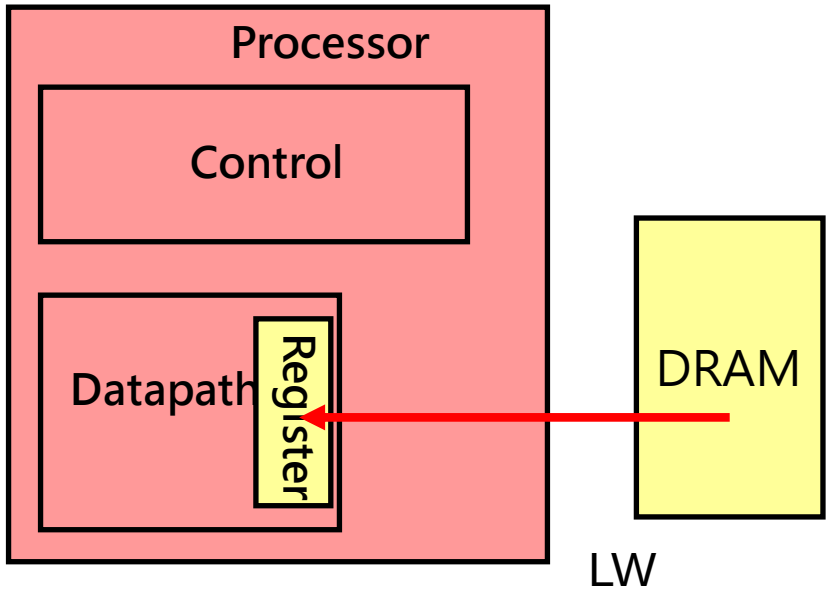


- Assume value g is in $s1, h is in $s2, value i is in $s3, and value j is in $s4 (Hint: use $t0 and $t1 to store temporary values)
- Store result f in $s0

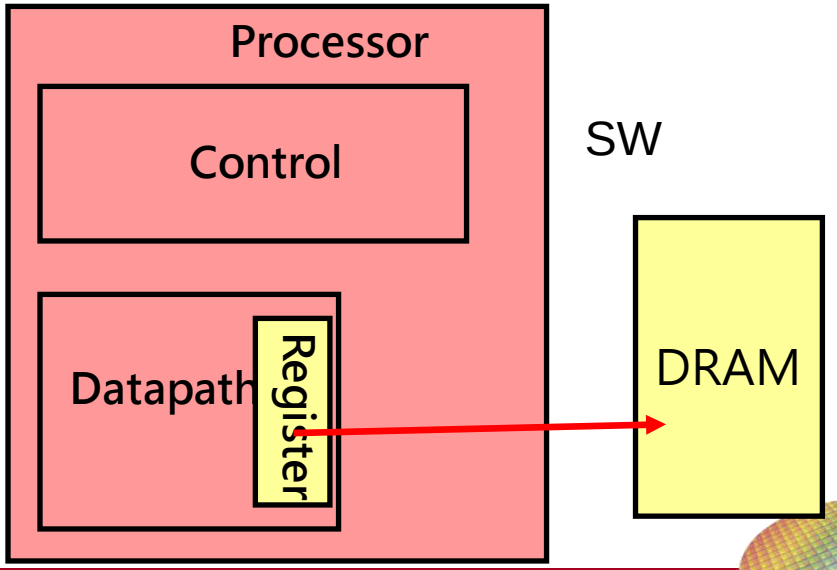
```
add $t0, $s1, $s2  # $t0 = g+h
add $t1, $s3, $s4  # $t1 = i+j
sub $s0, $t0, $t1  # f = (g+h) - (i+j)
```

Data Transfer (Memory Access) Instruction

- Data need to be load from memory into register before processing and written back to memory after processing
- Two instructions
  - Load Word: copy data from memory to register



Store Word : copy data from register back to memory

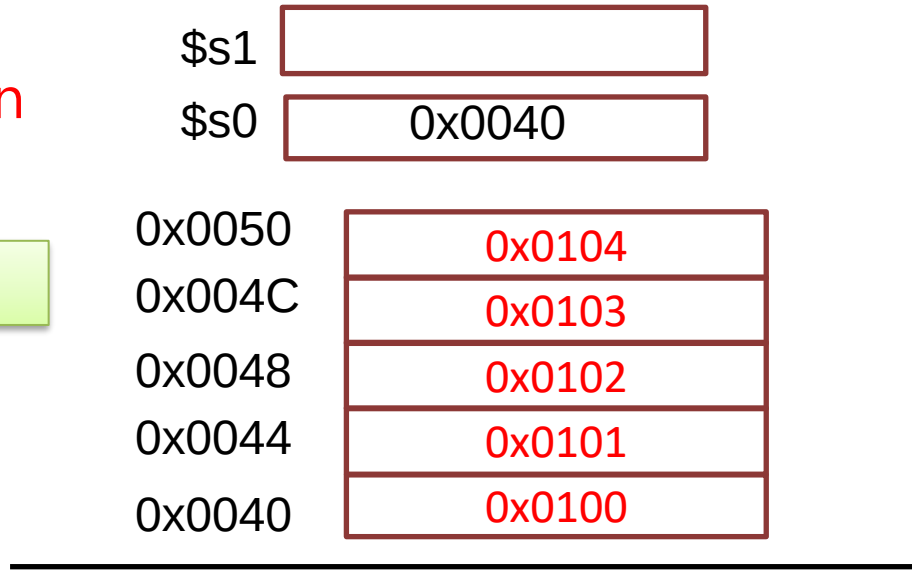


Load Word Instruction -LW

- lw (Load Instruction) loads data from the data memory through a memory address with an offset, to the destination register.

```
lw $rt, offset($rs) ➔  $rt= Memory[$rs+offset]
```

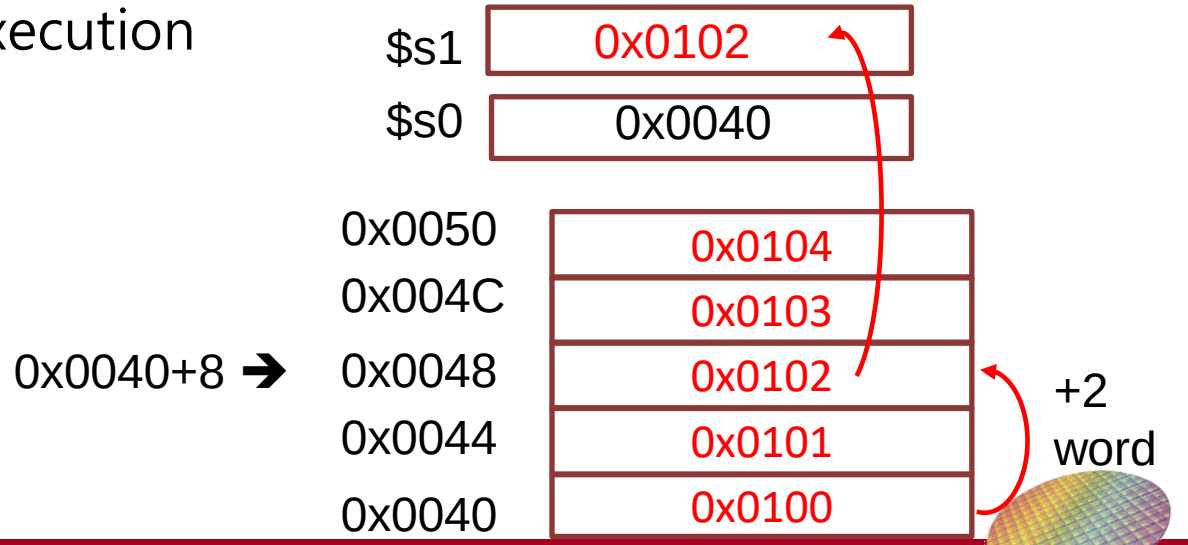
  – Copy a word from memory address "$rs+offset" into $rt
  – $rs is a register that contains a memory base address
  – offset is the distance
  – The data from memory is available in $rt after execution




```
lw $s1, 8($s0) #load word
```

Word offset

Base Address Register

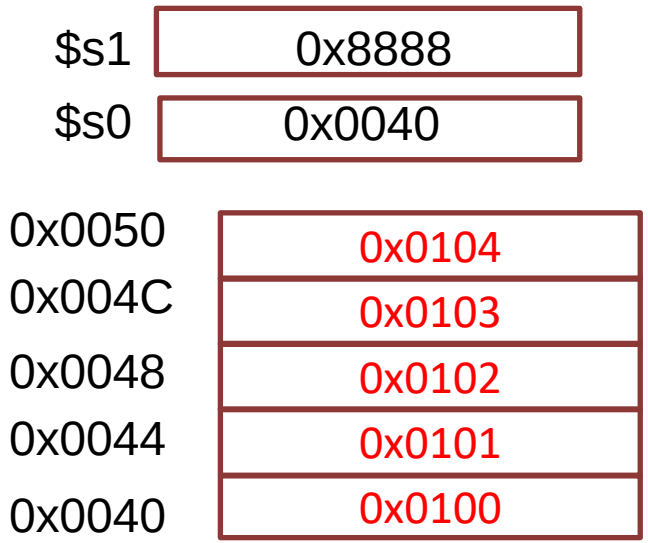


Store Word Instruction -SW

- sw (Store word) instruction stores data from a source register to a memory address on the data memory with an offset,

```
sw $rt, offset($rs) ➔  Memory[$rs+offset]= $rt
```

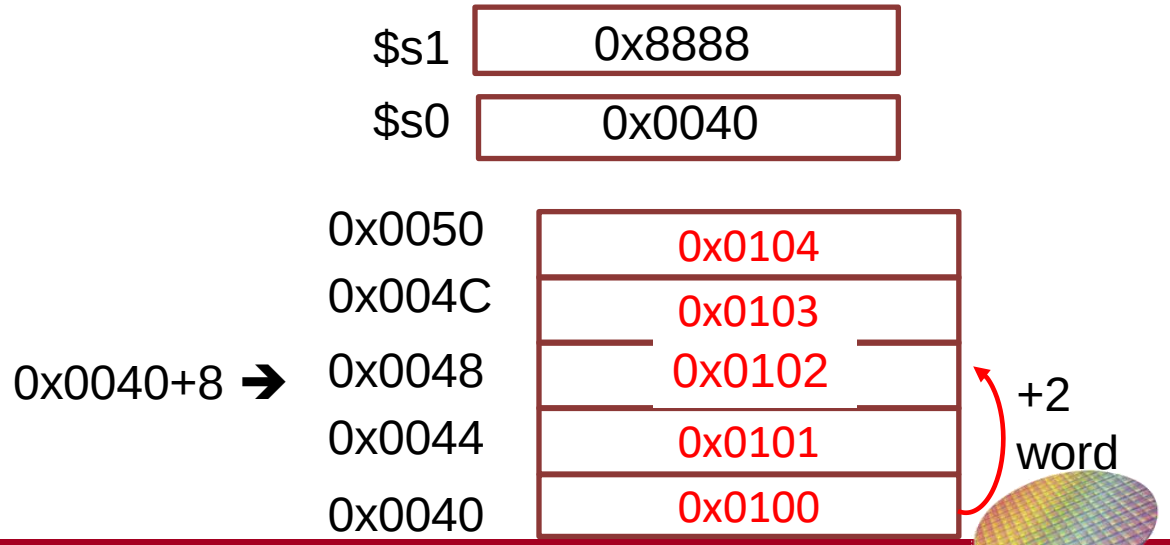
  – Copy a word from register $rt into memory address ($rs+offset)
  – $rs is a register that contains a memory base address
  – offset is the distance




```
sw $s1, 8($s0)
```

offset

Base Address Register



Memory Operand Example 1

- C code:

  `g = h + A[8];`

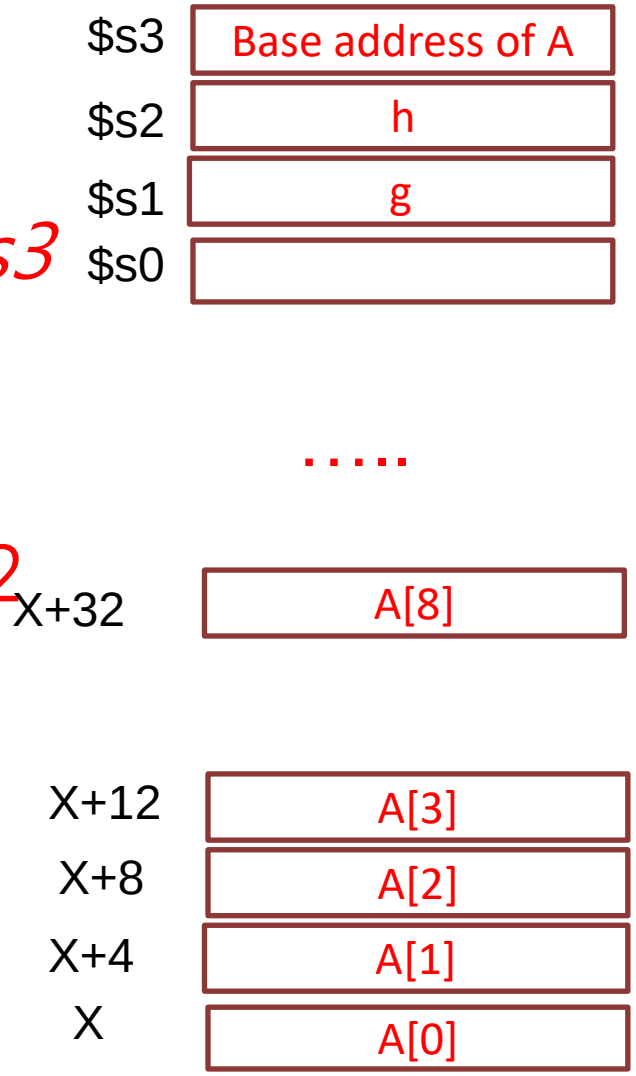
  – *g in $s1, h in $s2, base address of A in $s3*

- Convert the C code into MIPS code:
  - Each element in A array is 4 bytes
  - A[8] has index 8 and requires offset of 32

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

offset

base register



Logical Operations

- Recap: Basic logical operations includes
  - And, Or, Nand, Nor, Not, Xor

A	B	AND	OR	NAND	NOR	Not A	XOR
0	0	0	0	1	1	1	0
0	1	0	1	1	0	1	1
1	0	0	1	1	0	0	1
1	1	1	1	0	0	0	0

- Three instructions for logic operations are introduced
  - Bitwise AND: and
  - Bitwise OR: or
  - Bitwise NOR: nor
- These instructions performs bitwise operations (bit by bit)
- Useful for extracting and inserting groups of bits in a word

Logic Operations – and & or

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

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

$t2	0000 0000 0000 0000 0000 1101 1100 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	0000 0000 0000 0000 0000 1100 0000 0000

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

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

$t2	0000 0000 0000 0000 0000 1101 1100 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	0000 0000 0000 0000 0011 1101 1100 0000

Logic Operation -NOR Operations

- NOR operations

```
nor $t0,$t1,$t2➔$t0= ~($t1|$t2)
```

Registers

$t2	0000 0000 0000 0000 0000 0000 0000 1011
$t1	0000 0000 0000 0000 0000 0000 0000 0010
$t0	0000 0000 0000 0000 0000 0000 0000 0100

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS uses NOR instruction to implement NOT operations

```
not $t0,$t1 = nor $t0, $t1, $zero
```

$t1	0000 0000 0000 0000 0011 1100 0000 0000
~($t1\|$zero)	1111 1111 1111 1111 1100 0011 1111 1111
$t0	1111 1111 1111 1111 1100 0011 1111 1111

Logic Operation -Shift Left Logical

- sll (Shift left logical) instruction: Shifts a register value($rt) left by the shift amount listed in the instruction and places the result in a third register($rd)

```
sll $rd, $rt, imm ➔  $rd= $rt << imm
```

  - Shift left and fill with 0 bits
  - "sll by *i* bits" equal to multiplies by 2^i (00000011 << 2 ➔3 *2^2= 12 00001100)

```
sll $t0, $s0, 2  ➔ $t0= $s0 << 2bits
```

$s0	0000 0000 0000 0000 0000 1101 1100 0000
$t0	0000 0000 0000 0000 0000 0000 0000 0000

$t0	0000 0000 0000 0000 0011 0111 0000 0000

Logic Operation -Shift Right Logical

- srl (Shift right logical) instruction: Shifts a register value($rt) right by the shift amount listed in the instruction and places the result in a third register($rd)

```
sll $rd, $rt, imm ➔  $rd= $rt >> imm
```

  – Shift right and fill with 0 bits

```
srl $t0, $s0, 2  ➔ $t0= $s0 >> 2bits
```

$s0	0000 0000 0000 0000 0000 1101 1100 0000
$t0	0000 0000 0000 0000 0000 0000 0000 0000

⇨

$t0	0000 0000 0000 0000 0000 0011 0111 0000

Outline

- MIPS Assembly Instructions
  - Arithmetic Instruction
  - Data Transfer (Memory Access) Instruction
  - Logical Instruction
  - Conditional Branch
  - Unconditional Jump
- Instruction Encoding

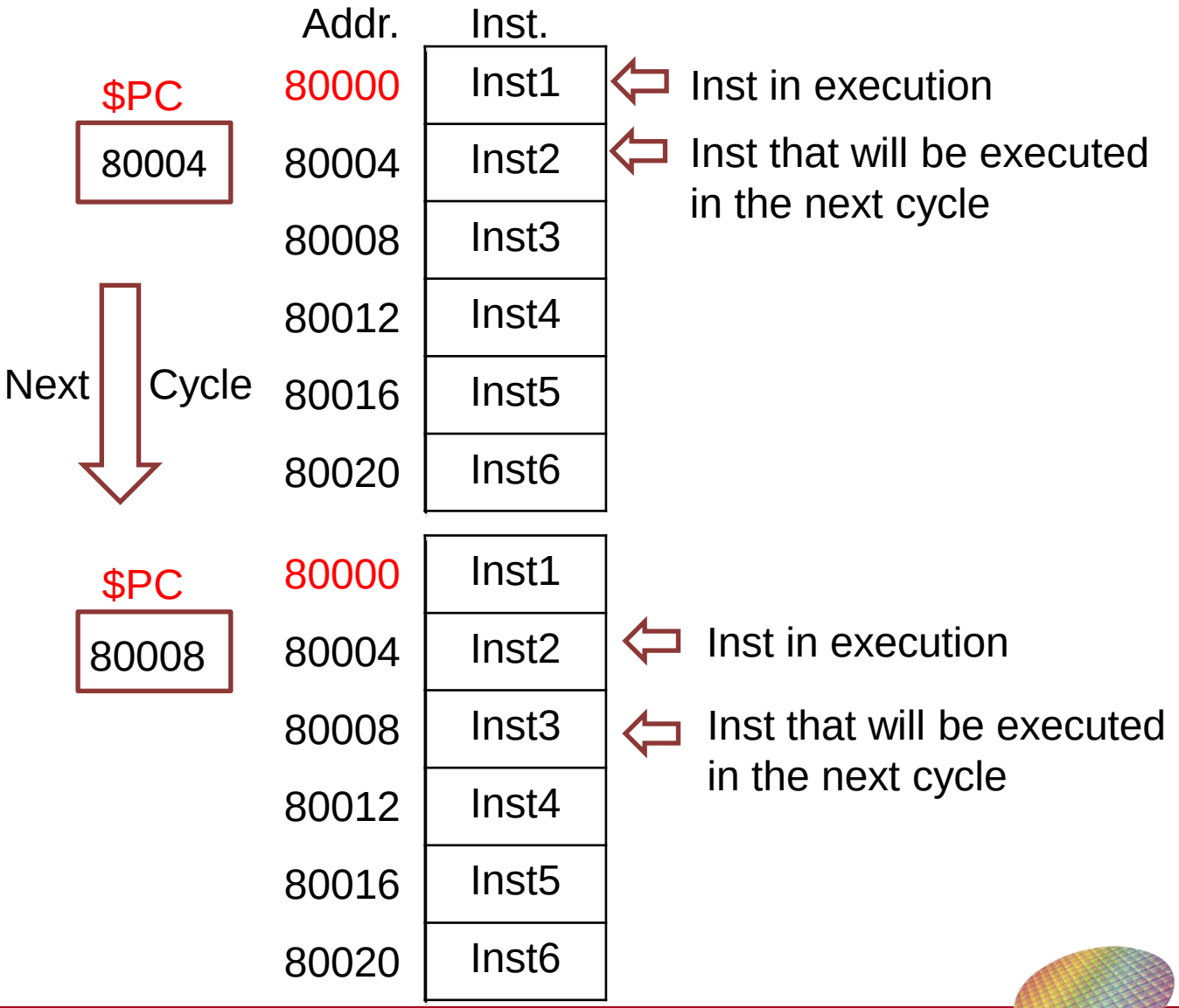
Program Counter

- A program counter is a special register that points to instructions
- Contains memory address (like a pointer) that points to the instructions being executed at the next cycle time.

- As each instruction gets fetched, the program counter increases its stored value by 4.

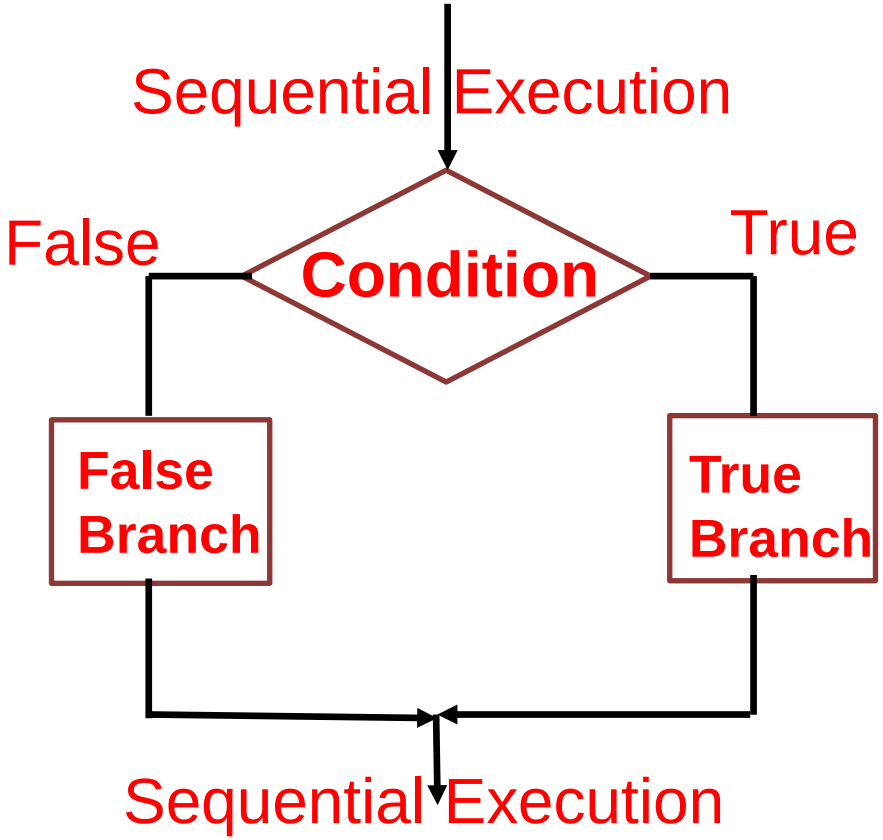
  PC = PC + 4

  (because the size of instruction is 4 bytes)

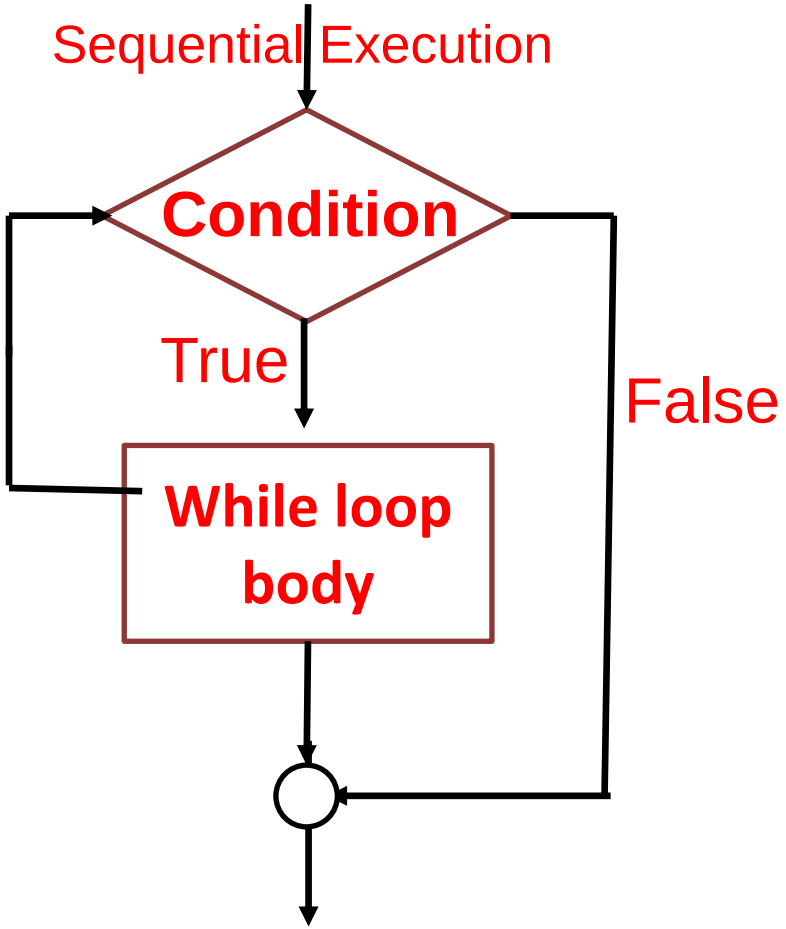


Instructions for Control operations – Condition Branch

- Control Instructions provide the ability alter their operation depending on data.
- Able to achieve alternative path control (if-then-else) or repeated path control (while)



True Branch

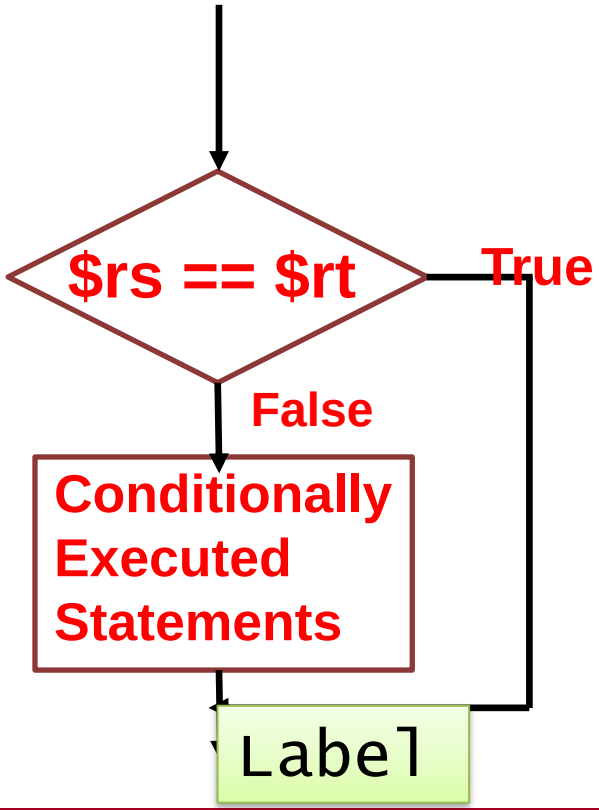


Conditional Branches -beq

- A conditional branch instruction branches to a new address only if a certain condition is met.
- The BEQ (branch on equal) instruction branches the PC if the first source register's contents and the second source register's contents are equal.

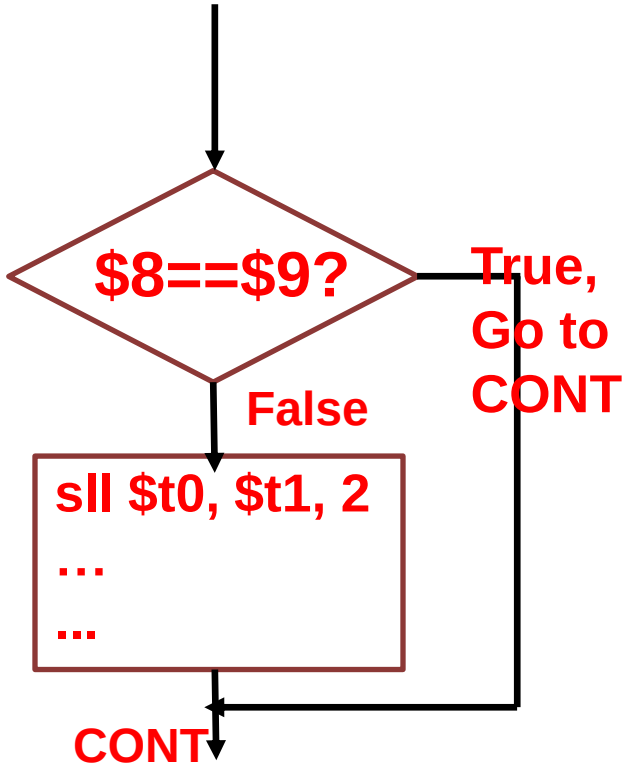
```
beq  $rs, $rt, Label
```

if register $rs == register $rt
goto Label
else
no effect.



Example: Conditional Execution

```
...                     # load values into $8 and $9
beq  $8,$9,cont         # branch if equal
sll  $t0,$t1, 2         #conditionally ….
….                      # executed statement
...                     #
cont:   add   $10,$10,$11  # always executed
```



Any instruction can be a target of a branch. The add instruction is here just as an example.)

Conditional Branches with PC-relative Addressing

- Conditional Branches use PC-relative Addressing to find Target Address

```
beq  $rs, $rt, Addr_Offset
```

if register $rs == register $rt
goto PC+4+ Addr_Offset*4
else
no effect.

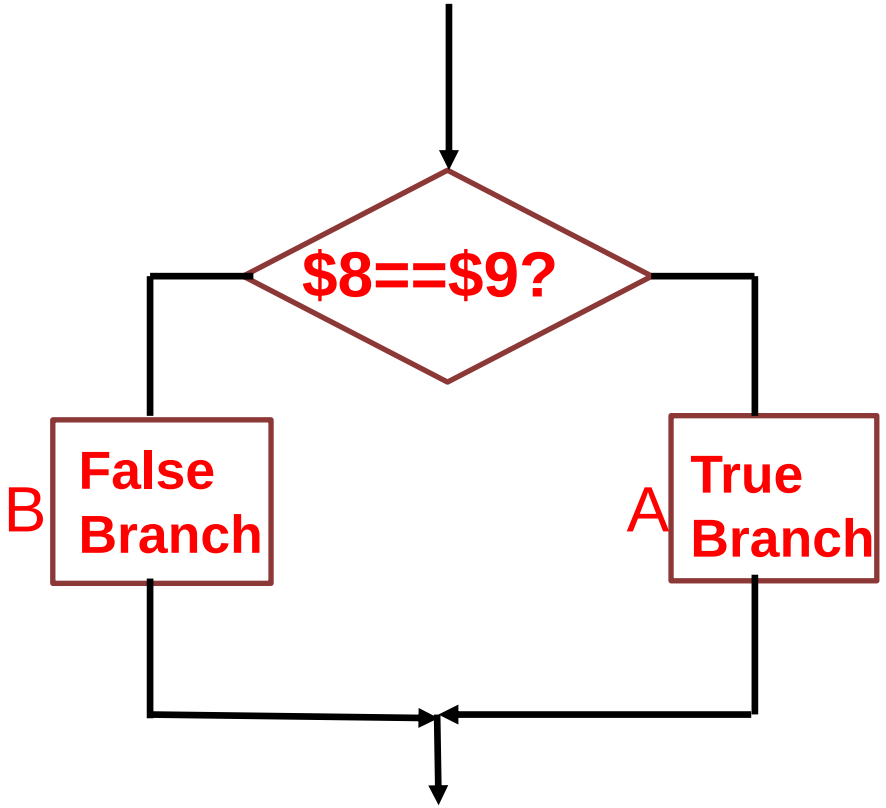
Suppose $s0=$t1,
(1) find target address of beq instruction
(2) the next instruction to be executed

Addr.	Inst.
04(0x04)	beq $s0 $t1 2
08(0x08)	add……
12(0x0C)	sub…..
16(0x10)	lw….
20(0x14)	sw….

Target address= 4+4+ 2*4=16
Next instruction to be executed: lw

Example: Two-way decision

```
        beq  $8,$9, equal   # branch if equal
        sll  $0,$0,0        #   false branch
        ...                 #  false branch
B       ...                 #
        j    cont           # jump to cont
equal:  ...
        ...                 #  true branch
A       ...                 #
cont:   add   $10,$10,$11  # always executed
```



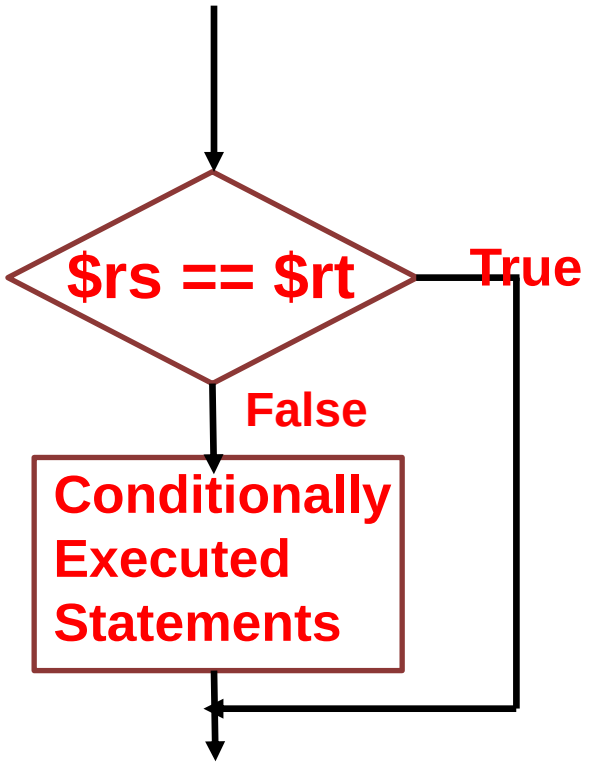
- If $8==$9 , execute A
- If $8 != $9 execute B
- A basic control structure is built out of small assembly instructions.

Conditional Branches -bne

- The BNE (branch on NOT-equal) instruction branches the PC if the first source register's contents and the second source register's contents are NOT equal.

```
bne   $rs, $rt, Label
```

if $rs != $rt
Goto Label
else
no effect.

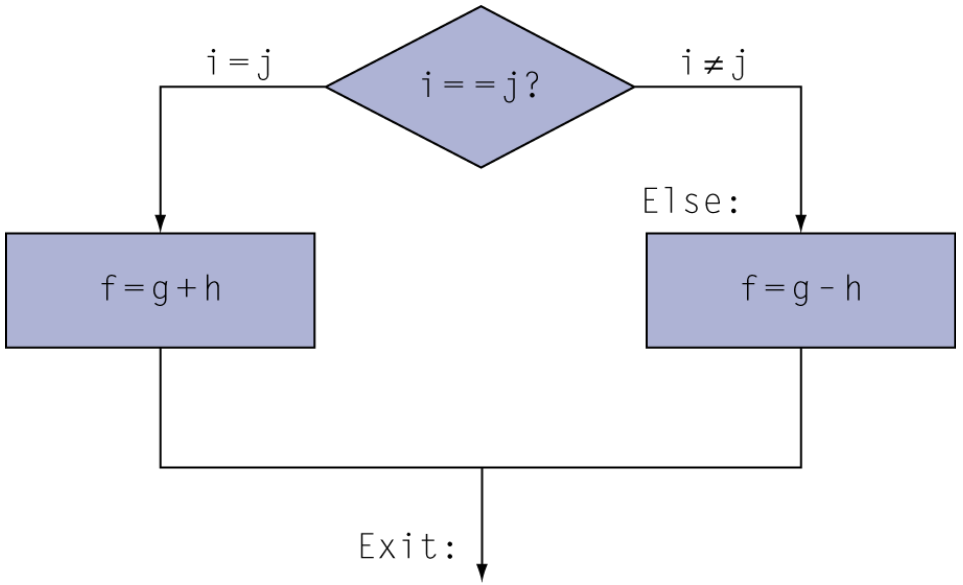


Compiling If Statements using bne

- Example: convert the C code into MIPS codes, assume variables f, g, h, i, j are stored in $s0~$s4

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

$s0	f
$s1	g
$s2	h
$s3	i
$s4	j



- Compiled MIPS code:

```
      bne $s3, $s4, Else   #if (i!=j) goto ELSE
      add $s0, $s1, $s2    # f= g+h
      j   Exit
Else: sub $s0, $s1, $s2    # f= g-h
Exit: …
```

```
      bne $s3, $s4, 2
      add $s0, $s1, $s2
      j   Exit
Else: sub $s0, $s1, $s2
Exit: …
```

Assembler calculates addresses

Conditional Operations: slt

- The SLT(set less than) instruction sets the $rd to the value 1 if the first source register's contents are less than the second source register's contents. Otherwise, it is set to the value 0.

```
slt  $rd,  $rs,  $rt
```

if register $rs < register $rt
$rd =1
else
$rd=0

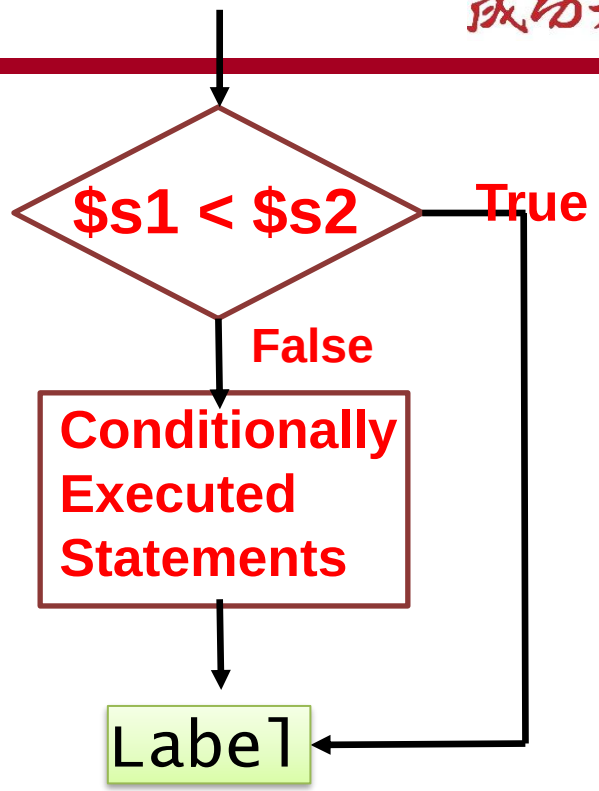
```
If $s0=1& $s1=2
After slt $t0, $s0, $s1
$t0 = ?  1
```

```
If $s0=1& $s1=1
After slt $t0, $s0, $s1
$t0 = ? 0
```

branch-if-less-than

- No branch-if-less-than in MIPS instruction
- How can we make a “psuedo” instruction

  “blt $s1, $s2, Label”?
- Combine “slt” and “bne”
  - “slt” decide if a register is less than another
  - “bne” decide execution path based on the result




```
slt $t0, $s1, $s2
```

⇨

```
if ($s1 < $s2) t0=1;
Else t0=0
```

```
if ($s1 < $s2)
branch to Label
```

⇨

```
slt $t0, $s1, $s2
bne $t0, $zero, Label
```

(Unconditional) Jump

- J (jump) instruction: jump the target address (Target address is a absolute address, and could be anywhere in text segment)
- Jump instruction: j

```
j  label ➔ Jump to Label
```

Or

```
j  "address" ➔ Jump to address*4 (PC= address*4)
```

- Address needs to times 4 to obtain byte address

```
j  600 ➔ Jump to 600*4=2400 (PC=2400)
```

Instruction Encoding

- Assembly instructions are into 32-bit binary machine code (aka instruction word)
- Normally done by Assembler
- The layout is called the instruction format

- Include Three Format: I-Format & R-Format & J-Format

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R:	opcode	rs	rt	rd	shamt	funct

				16 bits
I:	Opcode	rs	rt	Address/immediate

		26 bits
J:	Opcode	Target Address

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)
```

Assembler

Binary machine language program (for MIPS)

```
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

Recall: Register Name and Number

Name	Register number	Usage
$zero	0	The constant value 0
$v0–$v1	2–3	Values for results and expression evaluation
$a0–$a3	4–7	Arguments
$t0–$t7	8–15	Temporaries
$s0–$s7	16–23	Saved
$t8–$t9	24–25	More temporaries
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address

- Register that are frequently used

e.g.
$s1=$17,
$s2=$18,
$t0=$8

Instruction Opcode

Op(31:26)								
28-26 31-29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	Jump	Jump &link	Branch eq	Branch ne	blez	bgtz
1(001)	Add immediate	Addiu	Set less than imm.	Set less than imm. Unsigned	andi	ori	xori	Load upper imm.
2(010)	TLB	FlPt						
3(011)								
4(100)	Load byte	Load half	Lwl	Load word	Load byte unsigned	Load half unsigned	lwr	
5(101)	Store byte	Store half	Swl	Store word			swr	
6(110)	Load linked word	lwcl						
7(111)	Store cond. word	swcl						

Function in R-format instructions

Op(31:26)=000000 (R-format), funct(5:0)								
2-0 5-3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	Shift left logical (sll)		Shift right logical(srl)	sra	sllv		srlv	srav
1(001)	jump register	jalr			Syscall	Break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	Mult	Multu	div	Divu				
4(100)	Add	Addu	Subtract	Subu	And	Or	Xor	Nor
5(101)			Set l.t.	Set l.t. unsigned				
6(110)								
7(111)								

Funct for “add”: 100000, “and”: 100100, sll: 000000

MIPS R-format Instructions

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

- Suitable for instruction with 3 operands
- Instruction fields
  - op: operation code (opcode) => indicate the operation
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount ➔ used in the sll, srl instructions
  - funct: function code (extends opcode)=>select the specific variant of the operation in the op field

R-format Example (add, and, ..etc.)

```
add $rd, $rs, $rt➔$rd = $rs + $rt
```

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

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	$s1	$s2	$t0	0	add
				No shift	
0	17_{10}	18_{10}	8	0	32
000000	10001	10010	01000	00000	100000

Note:
$s1=r17
$s2=r18
$t0=r8

Encoded Machine code

$00000010001100100100000001000000_2 = 0x02324020_{16}$

Hex decimal format

Encode "and" and "sll" instructions

```
and $rd, $rs, $rt
```

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

				unused	
0	rs	rt	rd	shamt	100100_2
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
0	9	10	8	0	100100_2
000000	01001	01010	01000	00000	100100_2

Instruction format for and:
op:0, funct: 100100_2

```
sll $rd, $rt, imm ➔  $rd= $rt << imm
```

```
sll $t1, $t0, 2      # $t1= $t0 << 2bits
```

	unused				
0	rs	rt	rd	shamt	000000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
0	0	8	9	2	0

Instruction format for sll:
op:0, funct: 0
shamt: how many positions to shift

MIPS I-format Instructions (lw, sw, addi,…,etc.)

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

- load instructions
  - $rt: destination register number
  - Constant: -2^{15} to $+2^{15}-1$ because 4th field is 16-bit
  - Address: offset added to base address in $rs

```
lw $rt, offset($rs) ➔  $rt= Memory[$rs+offset]
```

Opcode: lw:35_{10} (or 100011_2)
Note: $t0:r8, $t1:r9

```
lw $t0, 16($t1)
```

op	rs	rt	Address offset
35	9	8	16
100011	01001	01000	0000000000010000

MIPS I-format Instructions - sw

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

- store instructions
  - $rt: source register number
  - Constant: -2^{15} to $+2^{15}-1$ because 4th field is 16-bit
  - Address: offset added to base address in $rs

```
sw $rt, offset($rs) ➔  Memory[$rs+offset]= $rt
```

```
sw $t0, 16($t1)
```

op	rs	rt	Address offset
43	9	8	16
101011	01001	01000	0000000000010000

Opcode sw:43_{10} (or 101011_2)

Note: $t0:r8, $t1:r9

MIPS I-format Instructions - addi

- addi performs an addition on both the source register and the immediate data, and stores the result in the destination register

```
addi $rt, $rs, imm➔$rt = $rs +imm
```

Opcode addi:8_{10} (or 001000_2)

Note: $t0:r8, $t1:r9

```
addi $t0, $t1,5
```

$t0=$t1+5

op	rs	rt	Constant or imm
8	9	8	5
001000	01001	01000	0000000000000101

MIPS I-format Instructions – beq or bne

```
beq  $rs, $rt, Addr_Offset
```

if register $rs == register $rt
goto PC+4+ Addr_Offset*4
else
no effect.

Opcode beq:4_{10} (or 000100_2)
Note: $t0:r8, $t1:r9

```
beq  $t0, $t1, 4
```

op	rs	rt	Address_Offset
6 bits	5 bits	5 bits	16 bits
8	8	9	~~5~~
000100	01000	01001	0000000000000100

Assume $t0=$t1, given the following code, what is the next instruction to be executed?

Addr.	Inst.
16(=0x10)	beq $t0 $t1 4
20(=0x14)	inst1 ….
24(=0x18)	inst2…..
28(=0x1C)	inst3
32(=0x20)	inst4
36(=0x24)	inst5
40(=0x28)	inst6

Ans:
The PC of the instruction is 16.
PC+4+4*4=36, which is inst5

MIPS J-format Instructions - Jump

- J (jump) instruction: jump the target address (Target address is a absolute address, and could be anywhere in text segment)

```
j  "address" ➔ Jump to address*4 (PC= address*4)
```

`j 16`

Opcode J:2_{10} (or 000010_2)

op	address
6 bits	26 bits

000010	00000000000000000000010000
6 bits	26 bits

Addressing Mode: the ways of specifying an operand or a memory address.

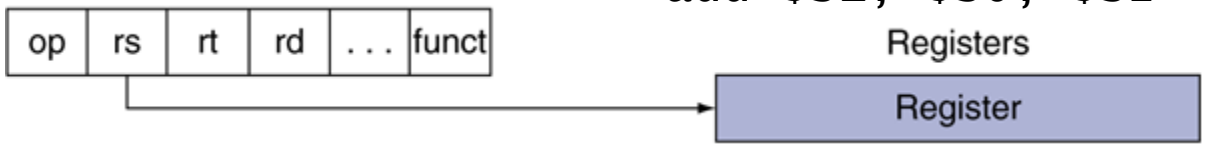
- Immediate addressing: operand is a constant within the instruction (e.g. addi)

- Register addressing: operand is a register (e.g

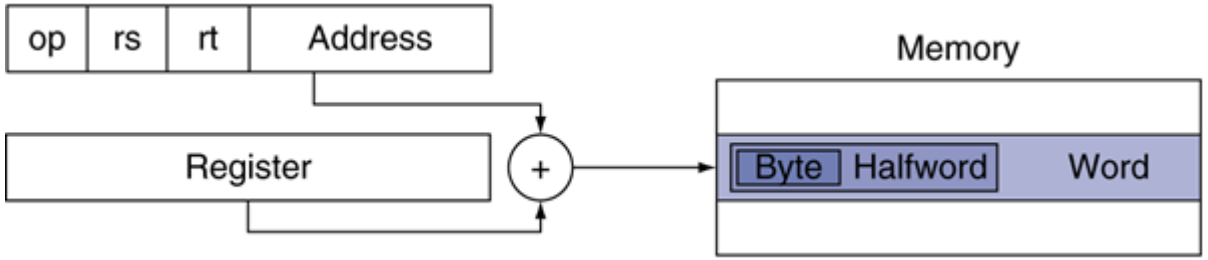
```
addi $s1, $s0, 1   # s1 = s0+1
```

- Base or displacement addressing: operand is at the memory location (e.g. lw, sw)

```
add $s1, $s0, $s2
```

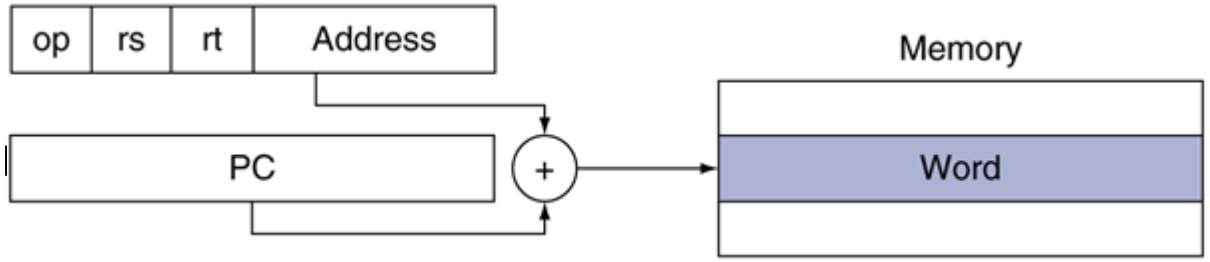



```
lw $t0, 32($s3)
```



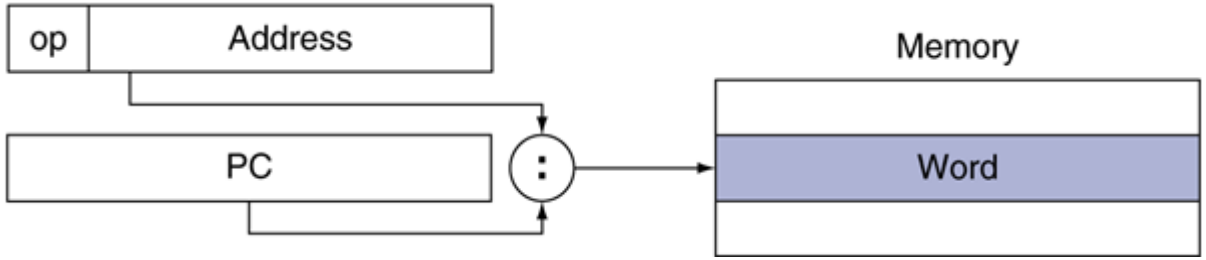
Summary: Addressing Mode (2)

- PC-relative addressing: branch address is the sum of PC and constant (e.g. beq)



- (Pseudo)direct addressing: ju

beq $s0,$s1, L1



Summary

- MIPS Assembly Instructions
  - Arithmetic Instruction
  - Data Transfer (Memory Access) Instruction
  - Logical Instruction
  - Conditional Branch
  - Unconditional Jump
- 3 Instruction Encoding Format
  - R type
  - I type
  - J type

Thank you for your attention