Chapter 6 :: Architecture

Quiz 3 → data & instructions
→ constant 0
→ 4

0 x 0
4
0
C

6.4-6.6 MIPS instructions

Digital Design and Computer Architecture

David Money Harris and Sarah L. Harris

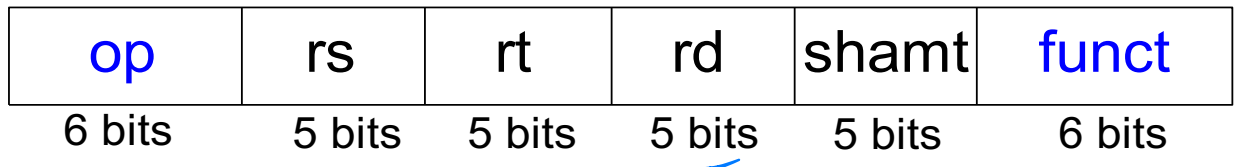
Machine Language

- Computers only understand 1's and 0's → CPU uses these to execute
- Machine language: binary representation of instructions
- 32-bit instructions
  - Again, simplicity favors regularity: 32-bit data and instructions
- Three instruction formats: → "what do the bits mean?"
  - R-Type: register operands
  - I-Type: immediate operand
  - J-Type: for jumping (we'll discuss later)

R-Type

- *Register-type*
- 3 register operands:
  - `rs`, `rt`: source registers
  - `rd`: destination register
- Other fields:
  - `op`: the *operation code* or *opcode* (0 for R-type instructions)
  - `funct`: the *function*

    together, the opcode and function tell the computer what operation to perform
  - `shamt`: the *shift amount* for shift instructions, otherwise it's 0



R-Type Examples

Assembly Code

```
add $s0, $s1, $s2
sub $t0, $t3, $t5
```

Field Values

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

op	rs	rt	rd	shamt	funct	
000000	10001	10010	10000	00000	100000	(0x02328020)
000000	01011	01101	01000	00000	100010	(0x016D4022)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Note the order of registers in the assembly code:

```
add rd, rs, rt
```

I-Type

- *Immediate-type*
- 3 operands:
  - `rs, rt:` register operands
  - `imm:` 16-bit two's complement immediate
- Other fields:
  - `op:` the opcode
  - Simplicity favors regularity: all instructions have opcode
  - Operation is completely determined by the opcode

I-Type

I-Type Examples

Assembly Code

```
addi $s0, $s1, 5
addi $t0, $s3, -12
lw   $t2, 32($0)
sw   $s1,  4($t1)
```

Field Values

op	rs	rt	imm
8	17	16	5
8	19	8	-12
35	0	10	32
43	9	17	4
6 bits	5 bits	5 bits	16 bits

Note the differing order of registers in the assembly and machine codes:

```
addi rt, rs, imm
lw   rt, imm(rs)
sw   rt, imm(rs)
```

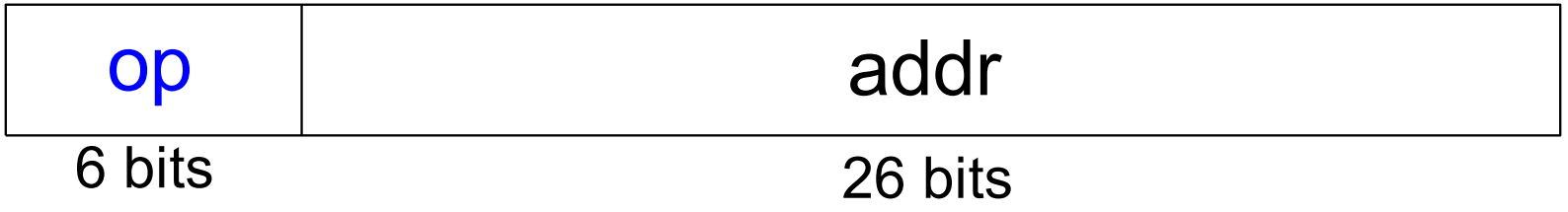
Machine Code

op	rs	rt	imm	
001000	10001	10000	0000 0000 0000 0101	(0x22300005)
001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
101011	01001	10001	0000 0000 0000 0100	(0xAD310004)
6 bits	5 bits	5 bits	16 bits	

Machine Language: J-Type

- *Jump-type*
- 26-bit address operand (`addr`)
- Used for jump instructions (`j`)

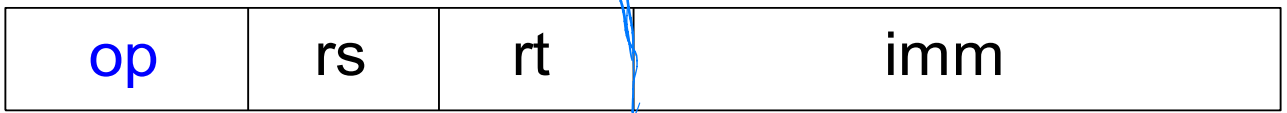
J-Type



Review: Instruction Formats

R-Type

I-Type

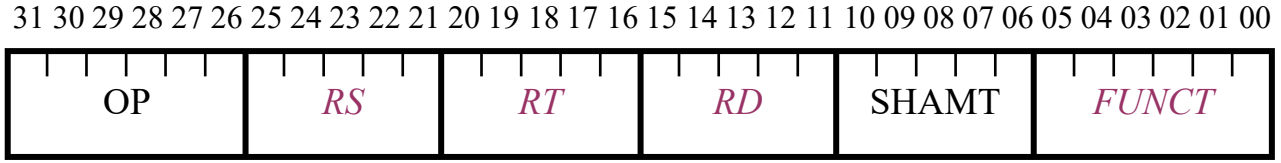


J-Type

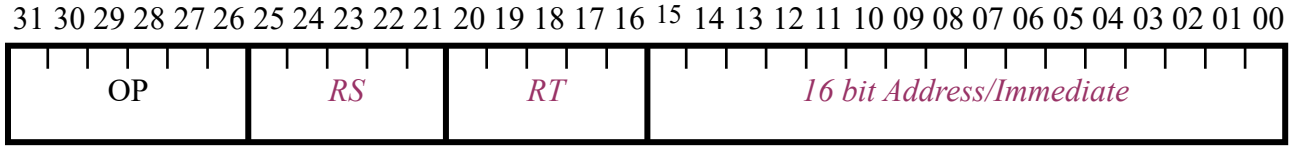
Bit number of different fields in instructions

- Opcode:31-26
- RS:25-21
- RT: 20-16
- RD(R type only):15-11
- FUNCT(R type): 5- 0
- Immediate (I type): 15- 0
- Address(J type): 25-0

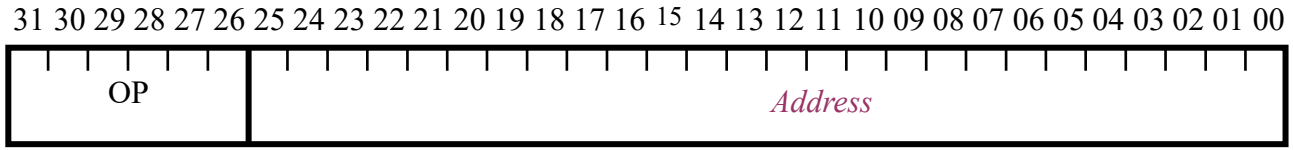
These will be used in single cycle processor in Ch.7



R type



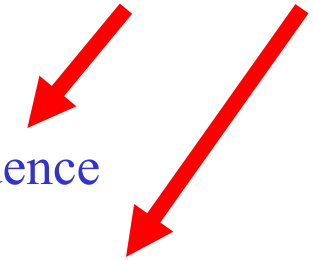
I type



The Power of the Stored Program

- 32-bit instructions and data stored in memory
- Sequence of instructions: only difference between two applications (for example, a text editor and a video game)
- To run a new program:
  - No rewiring required
  - Simply store new program in memory
- The processor hardware executes the program:
  - *fetches* (reads) the instructions from memory in sequence
  - performs the specified operation
- The program counter (PC) keeps track of the current instruction
- In MIPS, programs typically start at memory address 0x00400000

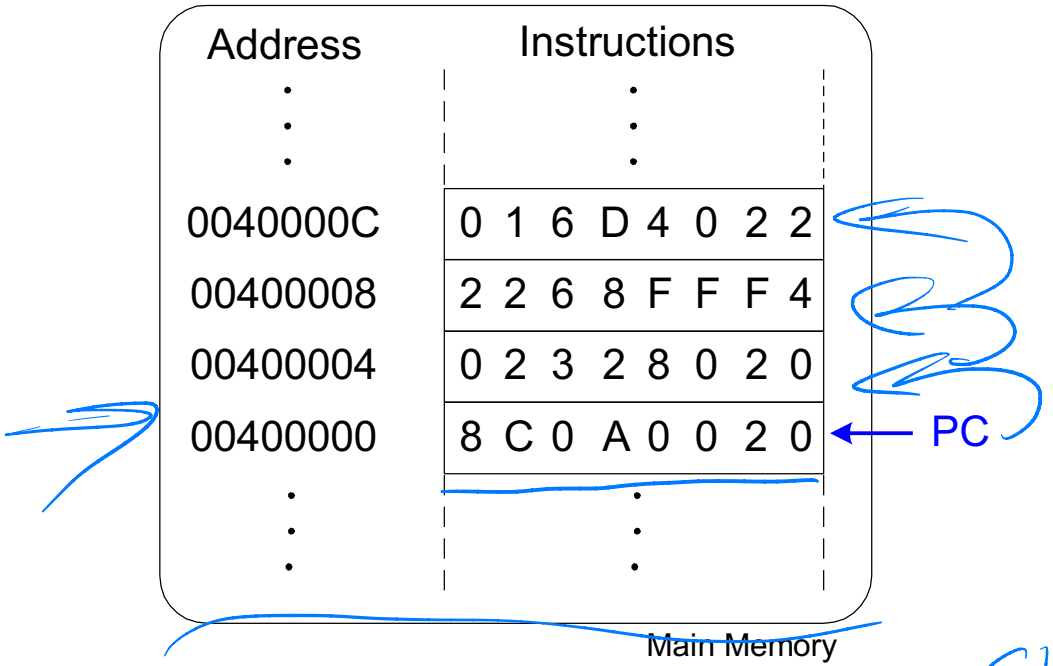
These will be used in single cycle processor in Ch.7



The Stored Program

Assembly Code	Machine Code
`lw $t2, 32($0)`	`0x8C0A0020`
`add $s0, $s1, $s2`	`0x02328020`
`addi $t0, $s3, -12`	`0x2268FFF4`
`sub $t0, $t3, $t5`	`0x016D4022`

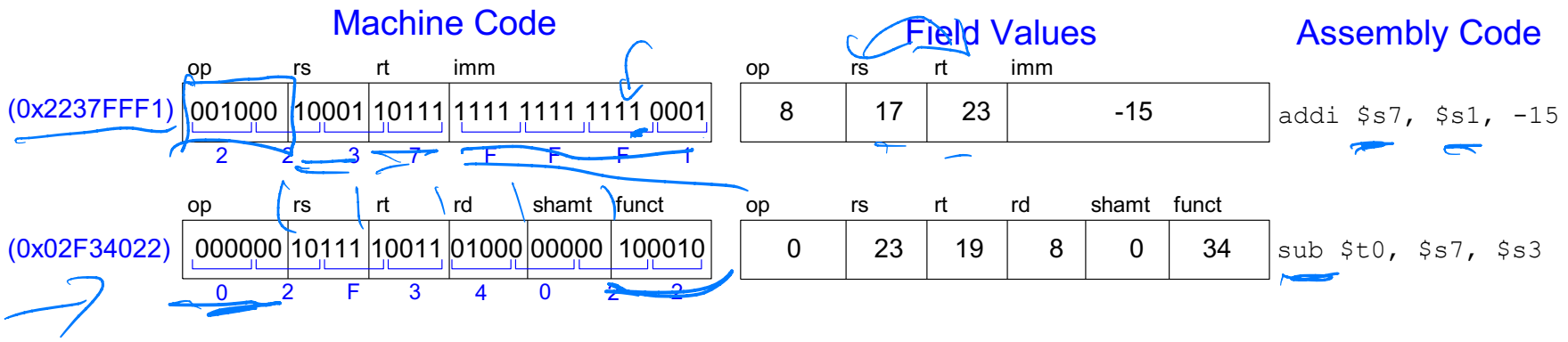
Stored Program



Stop here 9/15

Interpreting Machine Language Code

- Start with opcode
- Opcode tells how to parse the remaining bits
- If opcode is all 0's
  - R-type instruction
  - Function bits tell what instruction it is
- Otherwise
  - opcode tells what instruction it is



Programming

- High-level languages:
  - e.g., C, Java, Python
  - Written at higher level of abstraction
- Common high-level software constructs:
  - if/else statements
  - for loops
  - while loops
  - arrays
  - function calls

Logical Instructions

- **`and`, `or`, `xor`, `nor`**
  - `and`: useful for **masking** bits
    - Masking all but the least significant byte of a value:
      0xF234012F AND 0x000000FF = 0x0000002F
  - `or`: useful for **combining** bit fields
    - Combine 0xF2340000 with 0x000012BC:
      0xF2340000 OR 0x000012BC = 0xF23412BC
  - `nor`: useful for **inverting** bits:
    - A NOR $0 = NOT A
- **`andi`, `ori`, `xori`**
  - 16-bit immediate is zero-extended (*not* sign-extended)
  - `nori` not needed

Logical Instructions Example 1

Source Registers

$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

Assembly Code

```
and $s3, $s1, $s2
or  $s4, $s1, $s2
xor $s5, $s1, $s2
nor $s6, $s1, $s2
```

Result

$s3	0100	0110	1010	0001	0000	0000	0000	0000
$s4	1111	1111	1111	1111	1111	0000	1011	0111
$s5	1011	1001	0101	1110	1111	0000	1011	0111
$s6	0000	0000	0000	0000	0000	1111	0100	1000

$1 \oplus X = \overline{X}$

$0 \oplus X = X$

Logical Instructions Example 1

Source Registers

$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

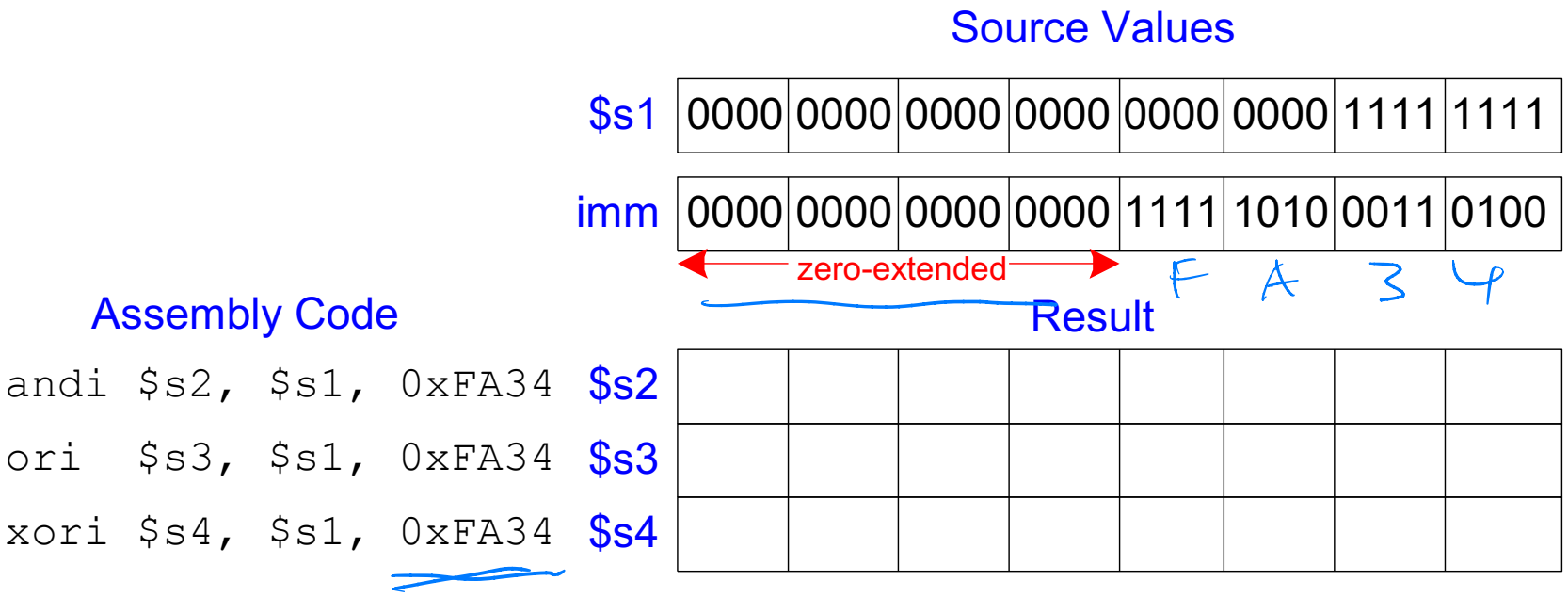
Assembly Code

```
and $s3, $s1, $s2
or  $s4, $s1, $s2
xor $s5, $s1, $s2
nor $s6, $s1, $s2
```

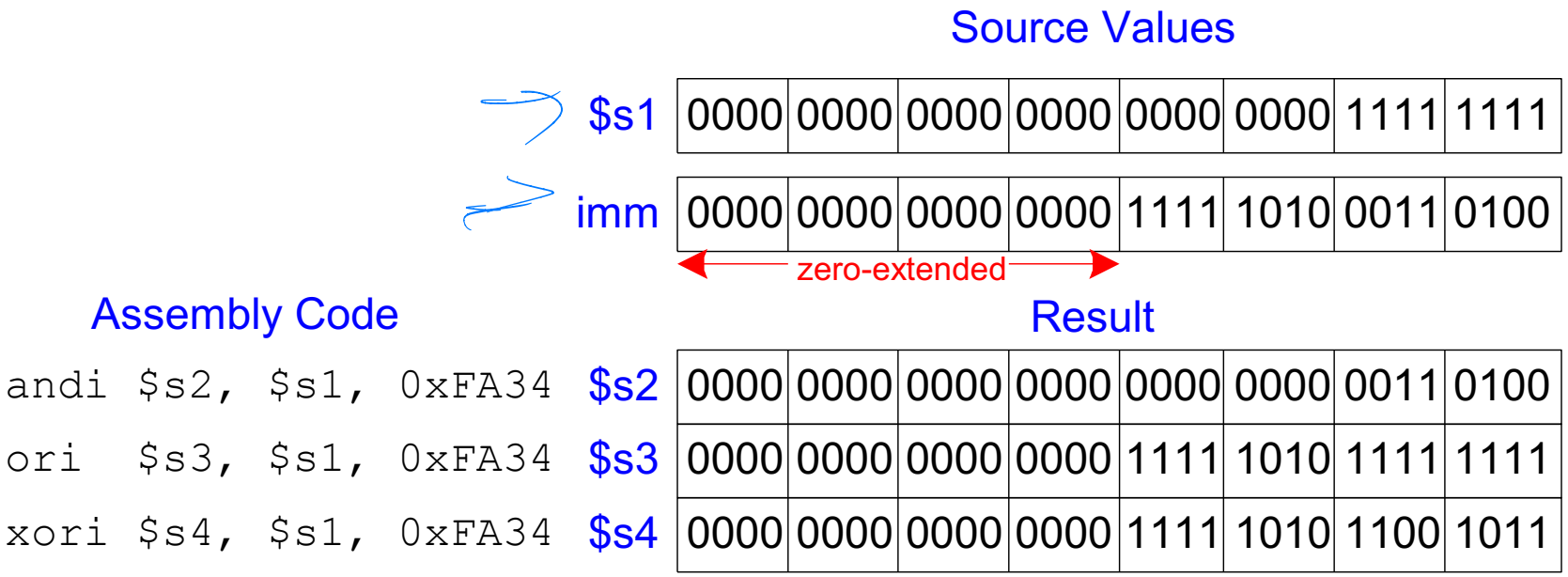
Result

$s3	0100	0110	1010	0001	0000	0000	0000	0000
$s4	1111	1111	1111	1111	1111	0000	1011	0111
$s5	1011	1001	0101	1110	1111	0000	1011	0111
$s6	0000	0000	0000	0000	0000	1111	0100	1000

Logical Instructions Example 2



Logical Instructions Example 2



Shift Instructions

- `sll`: shift left logical
  - **Example:** `sll $t0, $t1, 5  # $t0 <= $t1 << 5`
- `srl`: shift right logical
  - **Example:** `srl $t0, $t1, 5  # $t0 <= $t1 >> 5`
- `sra`: shift right arithmetic
  - **Example:** `sra $t0, $t1, 5  # $t0 <= $t1 >>> 5`

Shift Instructions

Assembly Code	op	rs	rt	rd	shamt	funct
`sll $t0, $s1, 2`	0	0	17	8	2	0
`srl $s2, $s1, 2`	0	0	17	18	2	2
`sra $s3, $s1, 2`	0	0	17	19	2	3
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Field Values

Machine Code

op	rs	rt	rd	shamt	funct	
000000	00000	10001	01000	00010	000000	(0x00114080)
000000	00000	10001	10010	00010	000010	(0x00119082)
000000	00000	10001	10011	00010	000011	(0x00119883)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Generating Constants

- 16-bit constants using `addi`:

C Code

```
// int is a 32-bit signed word
int a = 0x4f3c;
```

MIPS assembly code

```
# $s0 = a
addi $s0, $0, 0x4f3c
```

- 32-bit constants using load upper immediate (`lui`) and `ori`:

C Code

```
int a = 0xFEDC8765;
```

MIPS assembly code

```
# $s0 = a
lui $s0, 0xFEDC
ori $s0, $s0, 0x8765
```

Branching

- Execute instructions out of sequence
- Types of branches:
  - **Conditional**
    - branch if equal (`beq`)
    - branch if not equal (`bne`)
  - **Unconditional**
    - jump (`j`)
    - jump register (`jr`)
    - jump and link (`jal`)

(loops, if/else)
(functions)
function return
function calls

Conditional Branching (beq)

```
# MIPS assembly
addi $s0, $0, 4          # $s0 = 0 + 4 = 4
addi $s1, $0, 1          # $s1 = 0 + 1 = 1
sll  $s1, $s1, 2         # $s1 = 1 << 2 = 4
beq  $s0, $s1, target    # branch is taken
addi $s1, $s1, 1         # not executed
sub  $s1, $s1, $s0       # not executed

target:                  # label
 add  $s1, $s1, $s0      # $s1 = 4 + 4 = 8
```

Labels indicate instruction location. They can't be reserved words and must be followed by colon (:)

The Branch Not Taken (bne)

```
# MIPS assembly
    addi    $s0, $0, 4          # $s0 = 0 + 4 = 4
    addi    $s1, $0, 1          # $s1 = 0 + 1 = 1
    sll     $s1, $s1, 2         # $s1 = 1 << 2 = 4
    bne     $s0, $s1, target    # branch not taken
    addi    $s1, $s1, 1         # $s1 = 4 + 1 = 5
    sub     $s1, $s1, $s0       # $s1 = 5 - 4 = 1

target:
    add     $s1, $s1, $s0       # $s1 = 1 + 4 = 5
```

Unconditional Branching (j)

j-type

```
# MIPS assembly
    addi $s0, $0, 4              # $s0 = 4
    addi $s1, $0, 1              # $s1 = 1
    j    target                  # jump to target
    sra  $s1, $s1, 2             # not executed
    addi $s1, $s1, 1             # not executed
    sub  $s1, $s1, $s0           # not executed

target:
    add  $s1, $s1, $s0           # $s1 = 1 + 4 = 5
```

always jump