
http://xkcd.com/292/

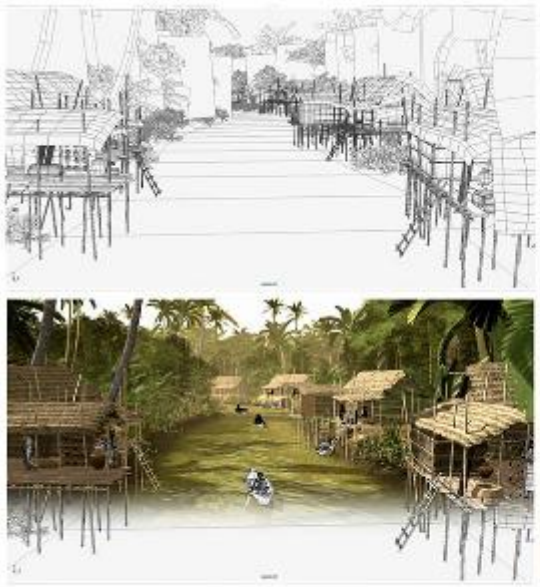
MONASH University

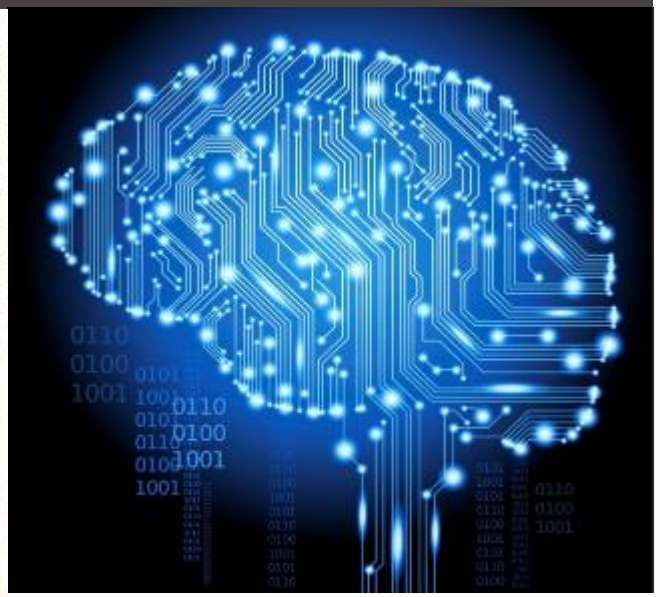
Information Technology

# FIT2085 Lectures 4 and 5

# Making decisions

Prepared by: M. Garcia de la Banda
based on D. Albrecht, J. Garcia





# Where are we up to:

- **The MIPS R2000 architecture**
  - 32 general purpose registers
  - Special purpose registers (HI, LO, PC, IR, etc)
  - ALU
  - Memory segments (text, data, heap, stack)
- **The fetch-decode-execute cycle**
- **The assembly language and assembler directives**
- **Part of the MIPS instruction set (maths, lw/sw, syscall)**

# Learning objectives for these 2 lectures

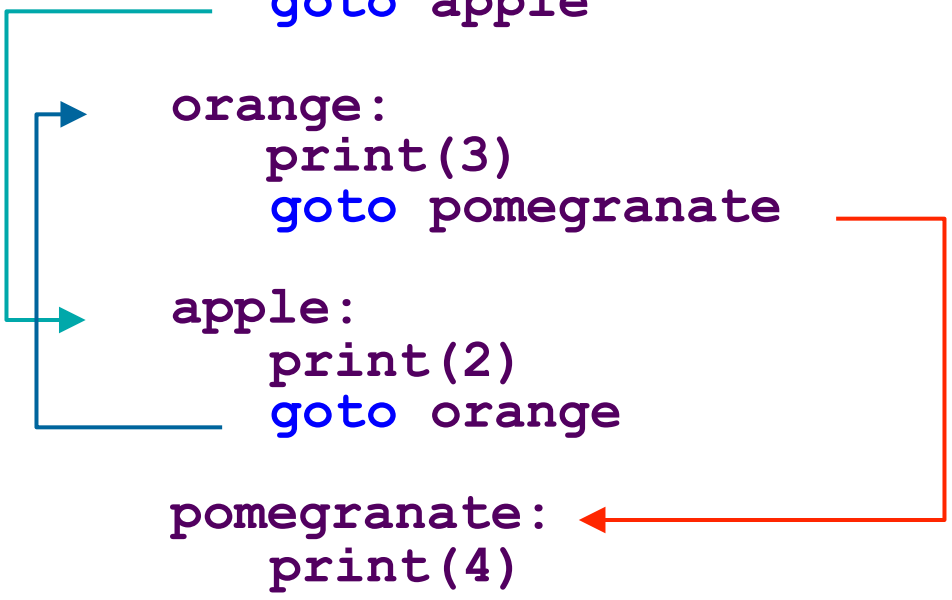
- To learn about MIPS selection and control transfer instructions
- To be able to use them to translate selection (if-else)
- To be able to use them to translate iteration (loops)
  - `while`
  - `for`
- To learn more about MIPS instruction format

# Blast from the past: the `goto` statement

- Remember: a label is an identifier for a program position
- The `goto` statement performs an unconditional jump to its label argument
- It promotes code whose control flow is extremely difficult to understand
- That is why it is not supported by most languages, including Python

# If Python had a `goto` statement …

```
# Code could be this ugly!
def main():

    print(1)
    goto apple

orange:
    print(3)
    goto pomegranate

apple:
    print(2)
    goto orange

pomegranate:
    print(4)
```



A very unclear way to print 1 2 3 4!!

https://crmbusiness.wordpress.com/2015/01/29/why-you-should-write-code-and-customizations-like-a-boy-scout/

# MIPS control transfer instructions

- **MIPS does have several kinds of goto instructions:**
  - They are called control transfer instructions
- **They are needed to change the control flow**
- **They replace the value in the Program Counter (PC) with the address value of a specified "destination"**
  - PC = destination address
- **Control is transferred to this new destination point at the next "fetch instruction" phase**
  - The next instruction loaded into IR is the one at the destination
- **They can be conditional and unconditional**

# Control Transfer Instructions

| | UNCONDITIONAL | | CONDITIONAL |
|---|---|---|---|
| | Register | Immediate | |
| **JUMP** | `jr` | `j` | |
| **CALL** | | `jal/jalr` | |
| **BRANCH** | | | `beq/bne` |

jump register

jump

branch equal/not-equal

jump and link/ register

# Jump and call instructions

- jump (go) to label

```
j foo          # set PC = foo
               # so, go to foo
```

- jump to address contained in register

```
jr $t0         # set PC=$t0, so go to the
               # address contained in $t0
```

- jump to label and link (remember origin)

```
jal foo        # $ra = PC+4; PC = foo, same as j foo
               # but setting a return address
```

- jump to register and link (remember origin)

```
jalr $t0       # $ra = PC+4; PC = $t0, same as
               # jr $t0 but setting return addres
```

Useful to call a function (return)

# Jump instructions – Example

```
                .text
1020   main:  addi $t0,$0,5
1024          addi $t1,$0,9
1028          addi $t2,$0,-3
102C          jal  subr
1030          add  $t3,$t3,$t3
1034          j    end
1038   subr:  add  $t3,$t1,$t0
103C          sub  $t3,$t3,$t2
1040          jr   $ra
1044    end:   …
```

$t0 ← 5

$t1 ← 9

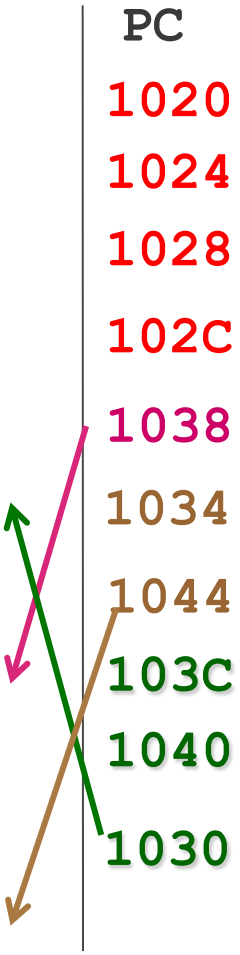
$t2 ← -3

$ra ←1030

$t3 ← 34

$t3 ← 14

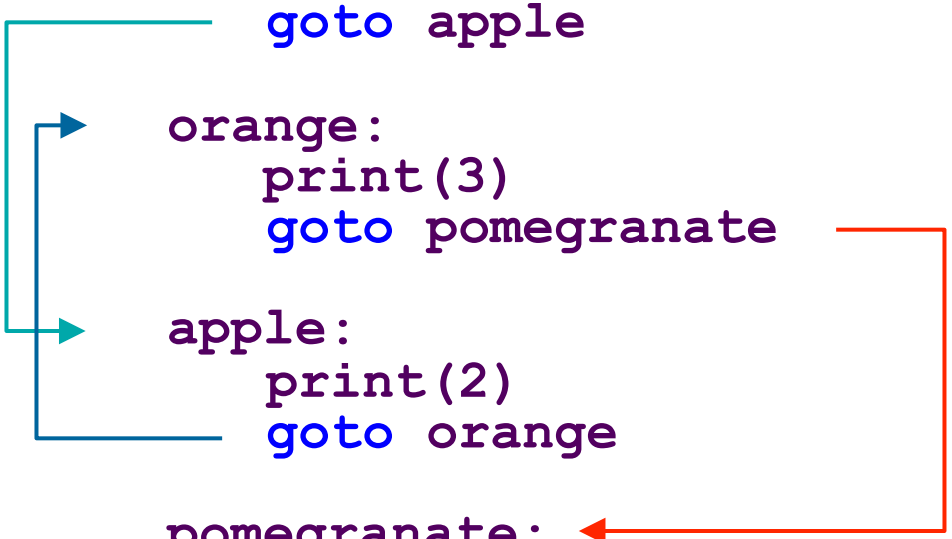
$t3 ← 17



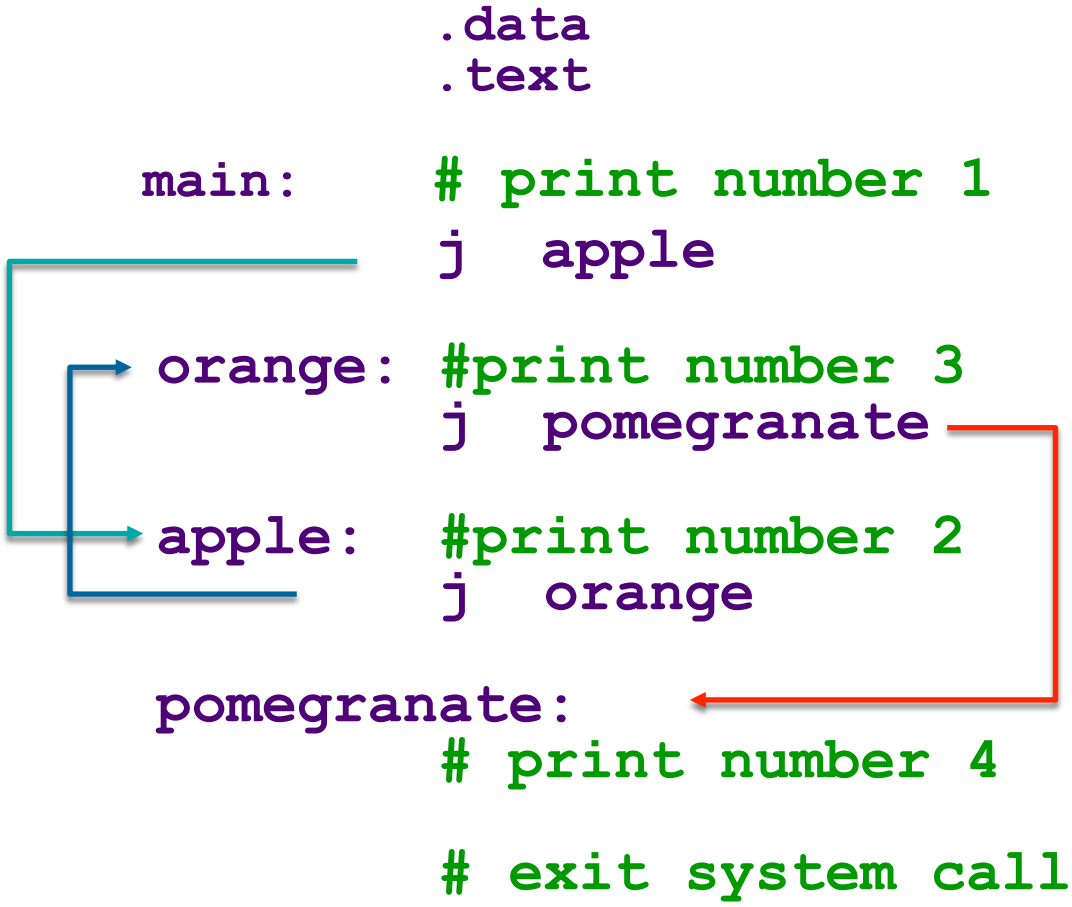
# Compiling selection

# Remember: If Python had `goto` …

```
# Code could be this ugly!
def main():

    print(1)
    goto apple

orange:
    print(3)
    goto pomegranate

apple:
    print(2)
    goto orange

pomegranate:
    print(4)
```



# Using MIPS `j` instruction



# Using MIPS j instruction

```
        .data


        .text
main:   # print number 1
        addi $v0, $0, 1 # print int
        addi $a0, $0, 1 # value 1
        syscall

        j  apple

orange: #print number 3
        addi $v0, $0, 1 # print int
        addi $a0, $0, 3 #value 3
        syscall

        j  pomegranate

# rest of program continued
# at right ...
```

```
# ... program continued from
# column at left.

apple:  #print number 2
        addi $v0, $0, 1 # print int
        addi $a0, $0, 2 # value 2
        syscall

        j  orange

pomegranate:
        # print number 4
        addi $v0, $0, 1 # print int
        addi $a0, $0, 4 # value 4
        syscall

        # exit system call
        addi  $v0, $0, 10 # exit
        syscall
```

Notice how the translation is “faithful”: each print puts 1 into $v0 (no reuse)

# Conditional branch instructions

- **Branch to a label if one value is equal/not equal another**
  - branch if equal to
    ```
    beq $t1, $t2, foo  # if $t1==$t2 goto foo
    ```
  - branch if not equal to
    ```
    bne $t1, $t2, foo  # if $t1!=$t2 goto foo
    ```
- **Note: `foo` is encoded as a signed offset (not as address)**
  - Counts in words and when added to PC points to address `foo`
- **If MIPS condition true, alter PC to equal label**
  - PC = PC + 4 + (sign extended immediate field <<2 )
- **If MIPS condition false, do normal PC update and continue at following instruction**
  - PC = PC + 4

What? See later…

# Conditional branch instructions

- **Is that it? Not really, MIPS also has:**
  - branch if less than

```
blt $t1, $t2, foo  # if $t1<$t2 goto foo
```

  - branch if less than or equal to

```
ble $t1, $t2, foo  # if $t1<=$t2 goto foo
```

  - branch if greater than

```
bgt $t1, $t2, foo  # if $t1>$t2 goto foo
```

  - branch if greater or equal to

```
bge $t1, $t2, foo  # if $t1>=$t2 goto foo
```

- **These are pseudoinstructions**
- **We will not use them (you must practice with basics)**
  - Only pseudoinstruction we will use in FIT2085 is `la`

# Control transfer is useful for selection

- **Selection is how programs make choices**
  - In Python: if, if-else, if-elif-else (like switch cases)
- **Achieved by selectively not executing some lines of code**

# Selection: if

```
# Sane people write
# code like this.

read(i)

if i < 0 :
    print(-5 * i)
```

Short way of saying
`i = int(input())`

```
# if Python had "goto" you
# could write it like this
# (ugh)

read(i)

if not i < 0:
    goto endif
    print(-5 * i)
endif:
```

Notice the negation of the condition

# Comparison Instructions

- **Control transfer is not enough, you also need to decide what to select: need to compare (`i < 0`)**

- **set less than**
  - `slt $t0,$t1,$t2 # if $t1<$t2 then $t0=1`
    `# else $t0 = 0`
  - Use this in conjunction with branch instructions to translate IF statements in high-level languages

- **set less than immediate**
  - `slti $t0,$t1,1  # if $t1<1 then $t0=1`
    `# else $t0 = 0`

- **Note: comparisons are performed by the ALU**
  - So comparison instructions are really arithmetic ones

# Set Less Than – Example

```
.text
addi $t1,$0,4          $t1 ← 4
addi $t2,$0,2          $t2 ← 2
slt $t0,$t1,$t2        $t0 ← 0
slti $t3,$t2,3         $t3 ← 1
```

# Practicing MIPS branching with `slt`

| $t0 | $t1 | X<Y $t2 | |
|---|---|---|---|
| X | Y | $t0< $t1 | $t1<$t0 |
| 10 | 15 | 1 | 0 |
| 15 | 15 | 0 | 0 |
| 15 | 10 | 0 | 1 → Y<X |

not X<Y
same as X>=Y

not Y< X
same as X <=Y

When translating:
always draw this table

# Putting it all together: if-then-else

- **Example: assume `X` is `$t0` and `Y` in `$t1`**
  - `if X == Y` ⇨
    ```
    bne $t0, $t1, endif
    ```
  - `if X < Y` ⇨
    ```
    slt $t2, $t0, $t1
    beq $t2, $0, endif
    ```
  - `if X <= Y` ⇨
    ```
    slt $t2, $t1, $t0
    bne $t2, $0, endif
    ```

Same as saying if not X==Y go to endif

Same as saying if not X<Y go to endif

Same as saying if Y<X go to endif which is equivalent: if not X<=Y go to endif

- **We use comparison to evaluate the condition (if needed)**
- **We use branch instructions to jump over the "then"**
- **We use jump instructions to jump over the "else"**

# If in MIPS

We will treat it as a global variable

```
i = 0
read(i)
if i < 0:
    print(-5 * i)
```

```
.data
i:   .word 0

.text
# Read integer "i" from input
addi  $v0, $0, 5      # system call code to
syscall               # read an int
sw  $v0, i            # store result in I

# Comparison part:
lw  $t0, i            # $t0=i
# if not i < 0: goto endif
slt $t1, $t0, $0      # $t1 = 0 if not i<0
beq  $t1, $0, endif   # if $t1 = 0 go to endif
# ... else fall through to here

# and print out -5*i
lw   $t0, i           # $t0=i
addi $t1, $0, -5      # store the 5 into a register
mult $t0, $t1         # -5*i
mflo $a0              # $a0 = -5*i
addi $v0, $0, 1       # call code to print an integer
syscall

endif:
addi  $v0, $0, 10     # call code to exit
syscall               # exit program
```

# Selection: if-else

```
i = 0

read(i)

if i < 0 :

    print(-5 * i)

else:

    print(5 * i)
```

```
i = 0

read(i)

if not i < 0:
    goto else

    print(-5 * i)

    goto endif

else:
    print(5 * i)

endif:
```

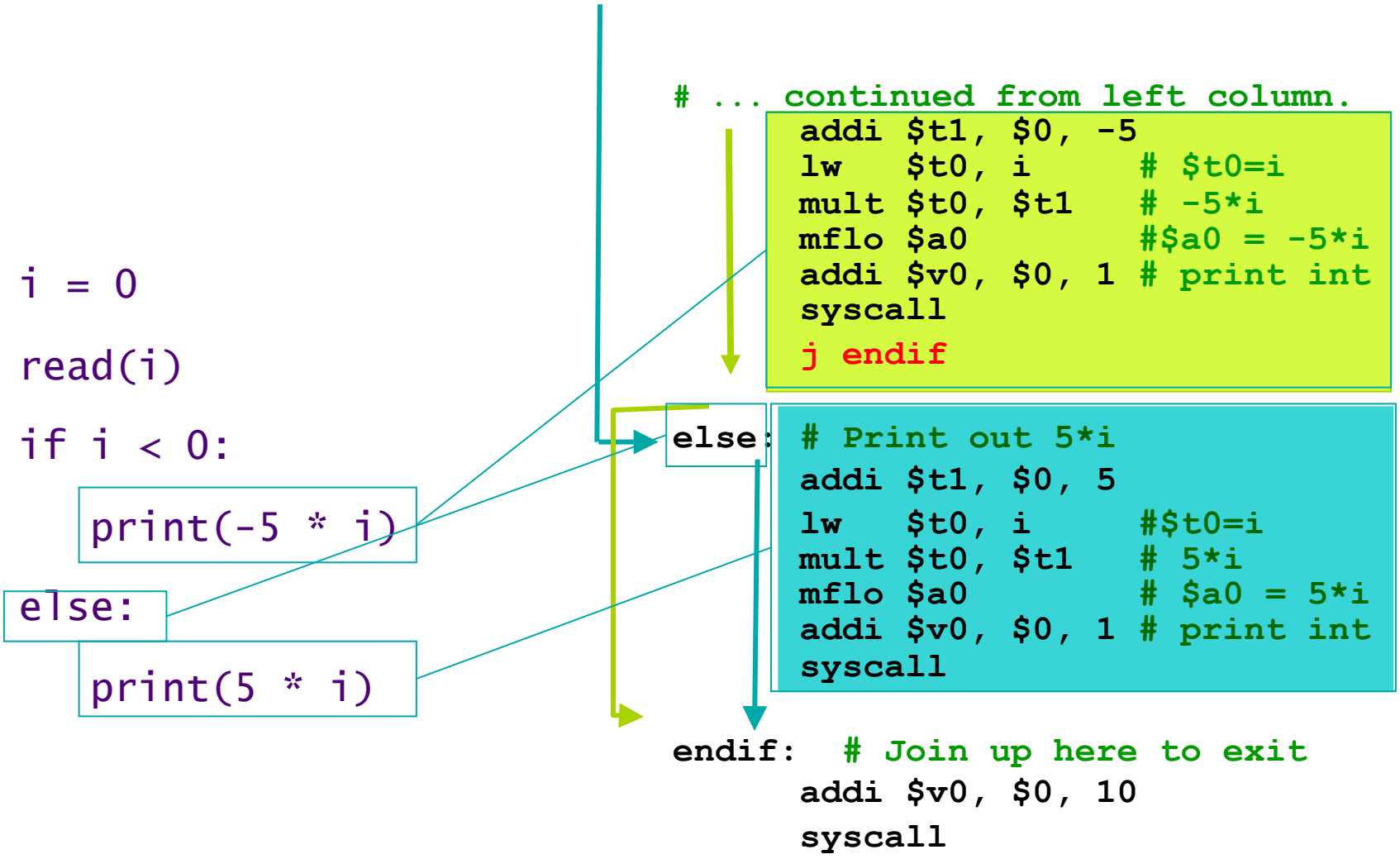
# if-else in MIPS

Again, we will treat it as a global variable

```
i = 0

read(i)

if i < 0:
    print(-5 * i)
else:
    print(5 * i)
```

```
.data
i: .word 0
.text
# Read i
addi  $v0, $0, 5    # read int
syscall
sw  $v0, i          # i = int
# Comparison part:
lw  $t0, i          # $t0 = i
# if not i < 0: goto else
slt $t1, $t0, $0    # $t1=0 if not i<0
beq $t1, $0, else   # if $t1=0 go to else

# fall through.
# and print out -5*i

# continued next column ...
```

# if-else in MIPS



# Your turn – what does this do?

```
            .text
1020  main: addi $t1,$0,5
1024        slti $t0,$t1,2
1028        beq  $t0,$0,foo
102c        addi $t1,$t1,1
1030        j    end
1034   foo: addi $t1,$t1,-1
1038   end:  …
```

$t1 ← 5

$t1 < 2? No, so: $t0←0

$t1 ← 4

What does it do?

If `$t1 < 2`, it adds 1 to `$t1`, otherwise it subtracts 1 from it

# Compiling iteration

# Reminder: Iteration

- **Iteration is the repetition of a section of code**
  - In Python, with **`while`**, **`for`**
  - **`while`** tests condition before loop entry
  - **`for`** is a shorthand for **`while`**
- **Achieved by sending control from the end of the loop back to the beginning**
  - Test some condition to prevent infinite loop

# Iteration: while

```
# while loops written
# like this ...

n = 0
f = 1

read(n)

# Compute factorial
while n > 0:
    f = f * n
    n = n- 1

print(f)
```

```
# ... could also be written
#  like this if Python had goto

    n = 0
    f = 1

    read(n)

    # Exit if condition=false
loop:
    if not n > 0:
        goto endloop
    # Body of loop.
    f = f * n
    n = n - 1
    # Repeat loop.
    goto loop
endloop:
    print(f)
```

Notice again the negation of the condition

# while in MIPS

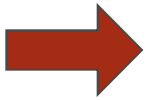
```
        .data
    n:  .word 0
    f:  .word 1

        .text
      # Read int, store in n
      addi $v0, $0, 5
      syscall
      sw  $v0, n

      # Now comes the loop.
      # Exit loop if not n > 0
loop: lw  $t0, n          #t0=n
      slt $t1, $0, $t0
      beq $t1, $0, endloop
      # ... else fall through.

# Continued at right ...
```

```
# ... Continued from left
      # Body of while loop.
      # f = f * n
      lw  $t0, f      # $t0=f
      lw  $t1, n      # $t1=n
      mult $t0, $t1 # f*n
      mflo $t0        # $t0=f*n
      sw  $t0, f      # f = f*n
      # n-- (n = n - 1)
      lw  $t0, n      # $t0=n
      addi $t0, $t0, -1
      sw  $t0, n      # n=n-1

      # End of loop, go back
      j  loop

endloop:# Print integer f
      addi $v0, $0, 1
      lw  $a0, f
      syscall

      addi $v0, $0, 10 # exit
      syscall
```

# Iteration: `for`

- **A `for` loop is essentially a simpler version of a `while` loop:**
  - Initialization, condition and increment code all in one place
- **To translate a `for` loop into MIPS, write it as a `while` loop**

```
for i in
  range(init, cond, inc):

    body
```



```
i = init
while (cond):

    body
    inc
```

# Iteration: for

```
# turn this for loop...

n = 0
read(n)

# Print n 9 times
for counter in
        range( 10,
               1,
               -1 ):

    print(n)
```

```
# into this while loop

n = 0
read(n)

# Print n 9 times
counter = 10
while counter != 1:

    print(n)
    counter -= 1
```

We will treat it as a global

# for in MIPS

```
        .data

      n: .word 0
counter: .word 10

        .text

        # Read integer, store in n
        addi $v0, $0, 5
        syscall             # read int
        sw $v0, n           # n=int

loop:   # For loop: condition.
        lw $t0, counter # counter
        # End when counter == 1.
        addi $t1, $0, 1
        beq $t0, $t1, end

# Continued at right ...
```

```
# ... Continued from left
# For loop: body.
addi $v0, $0, 1#print int
lw $a0, n       #n
syscall

# For loop: decrement.
lw $t0, counter
addi $t0, $t0, -1
sw $t0, counter

# Return to loop start.
j loop

end:    # Program finishes
addi $v0, $0, 10 # exit
syscall
```

# Summary

- **MIPS control transfer instructions**
- **MIPS comparison instructions**
- **Selection**
  - `if-else`
- **Iteration (loops)**
  - `while`
  - `for`

# More about MIPS instruction formats

# MIPS Instruction Format

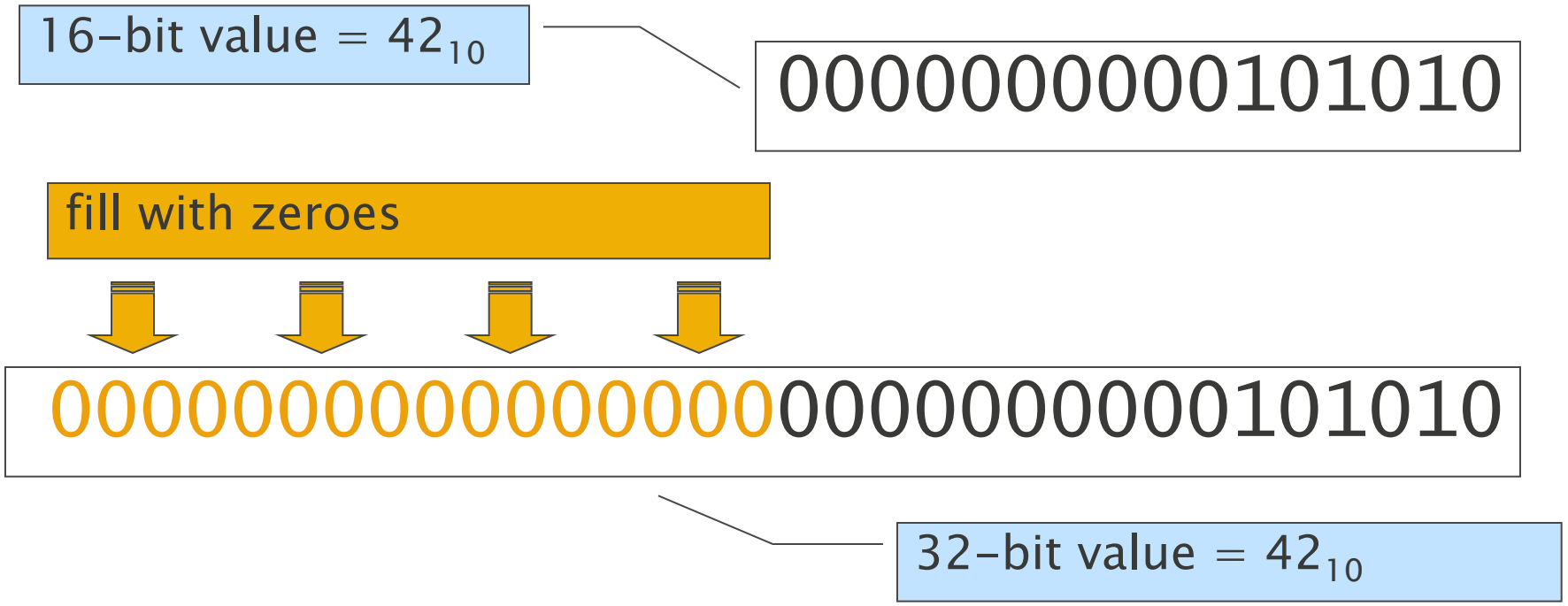
- **Remember: every MIPS instruction is 32-bits in size and occupies 4 bytes of memory**
- **Remember: each instruction contains**
  - opcode
    - operation code: specifies type of instruction
  - operands
    - values or location to perform operation on
      - registers
      - immediate (constant) numbers
      - labels (addresses of other lines of program)

# Dealing with immediate values

- **We have seen many instructions with immediate forms**
  - `addi`, `andi`, `ori`, `xori`, `sll`, `srl`, `sra`, `slti`
  - e.g., `addi $t0, $t1, 123` and `srl $t0, $t1, 5`
- **Immediate values have to be put in 32-bit internal registers so that they can go into the ALU**
- **But only 16 bits will fit into an instruction once you've encoded the opcode and two registers**
- **So how do we make a 16 bit value into a 32-bit value?**
  - It depends on the meaning of the immediate
  - In turn, this depends on the actual instruction

# Zero Extension

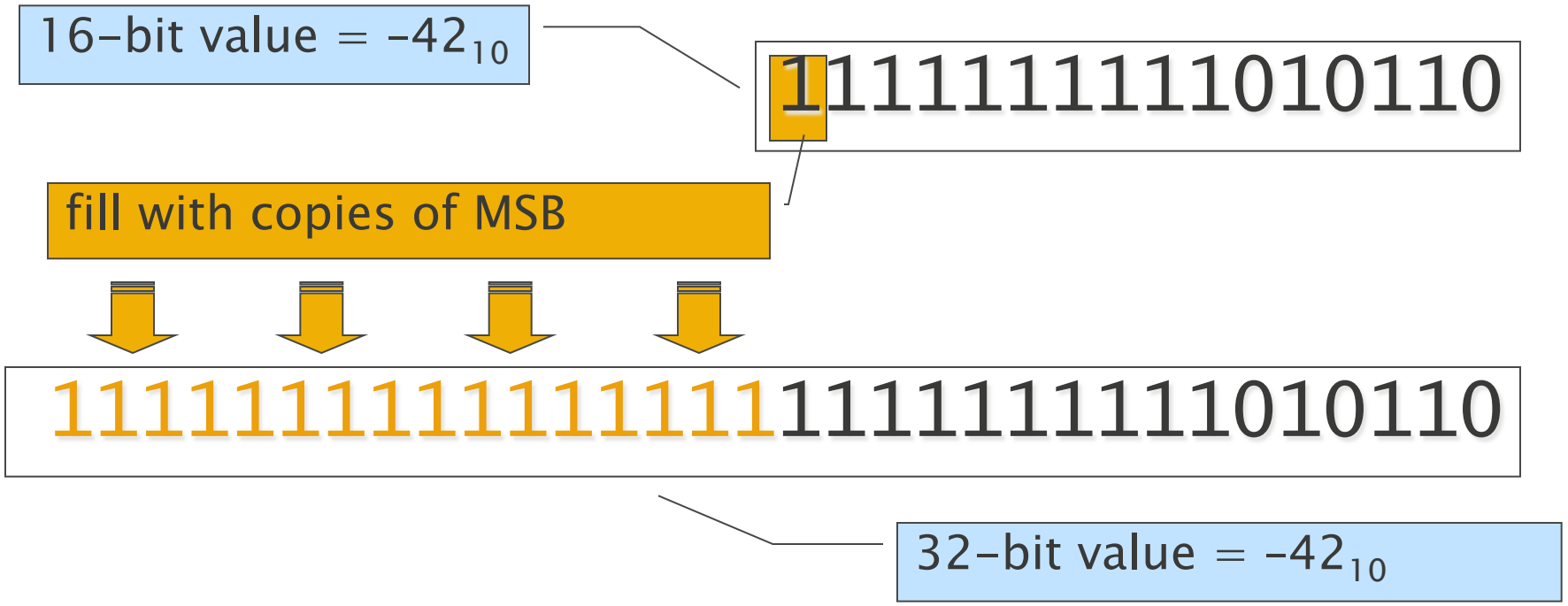
- **Used by** `andi,ori,xori`
- **Zeroes are added in front**



# Sign Extension

This is one of the main differences between **`addi`** (sign extended) and **`ori`** (zero extended)

- **Used for `addi,sll,srl,sra,slti`**
- **Sign is extended by duplicating MSB (sign bit)**



# MIPS Instruction Format

- **Three general assembler formats**

```
sub $t0, $t1, $t2
```

R (for "register") format instruction: three registers

subtract the contents of register $t2 from the contents of register $t1; put the result in register $t0

---

```
addi $v0, $a2, 742
```

I (for "immediate") format instruction: two registers and one immediate operand

add the immediate number 742 with the contents of register $a2; put the result in register $v0

---

```
j foo
```

J (for "jump") format instruction: has a line label (an address) as its only operand

jump (go) to the line with the label foo and continue running from there

# MIPS Instruction Format (cont'd)

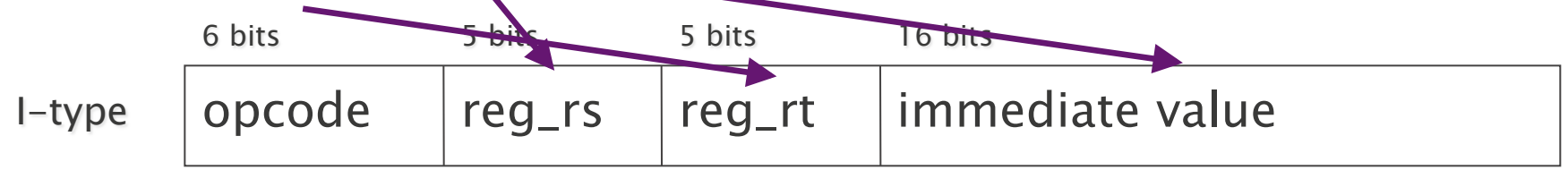
- **Three general assembler formats**

```
sub $t0, $t1, $t2
```

R (for "register") format instruction: three registers

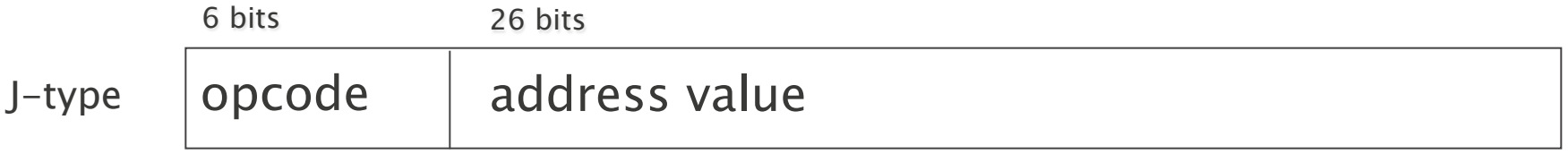

```
addi $v0, $a2, 742
```

I (for "immediate") format instruction: two registers and one immediate operand




```
j foo
```

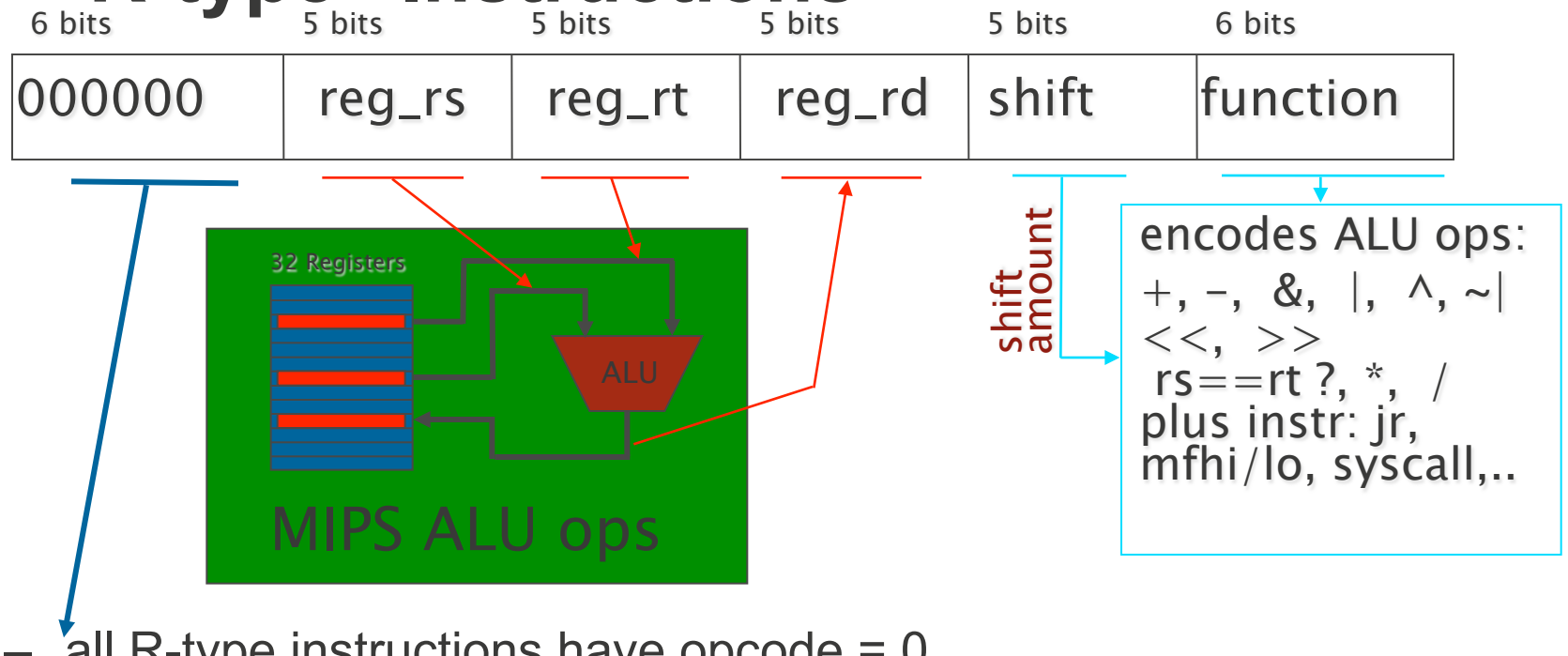
J (for "jump") format instruction: has a line label (an address) as its only operand



# "J-type" instructions

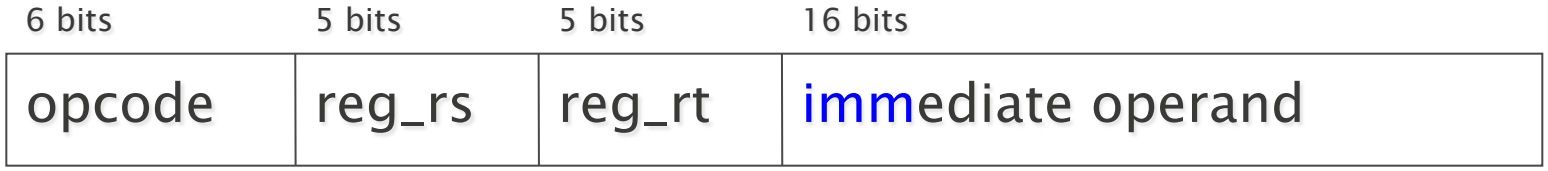

- **Only two MIPS instructions are j-type:**
  **`j label` and `jal label2` (opcodes 2 & 3, respectively)**
  - all label addresses in text segment are multiples of 4
  - assembler encodes 26 bit operand = (label addr >> 2)
  - this encoding improves maximum range of a jump
  - during instruction execution the operand is shifted << 2 and used to replace lowest 28 bits of the PC
  - `jal` is used exclusively for function calls
    - before PC is modified, PC+4 (=address of instruction after jal) is saved in register $ra ( return address )

# "R-type" Instructions




- all R-type instructions have opcode = 0
- they use register operands exclusively
- 6-bit function field specifies exact action
  - 28 different instructions encoded by this field

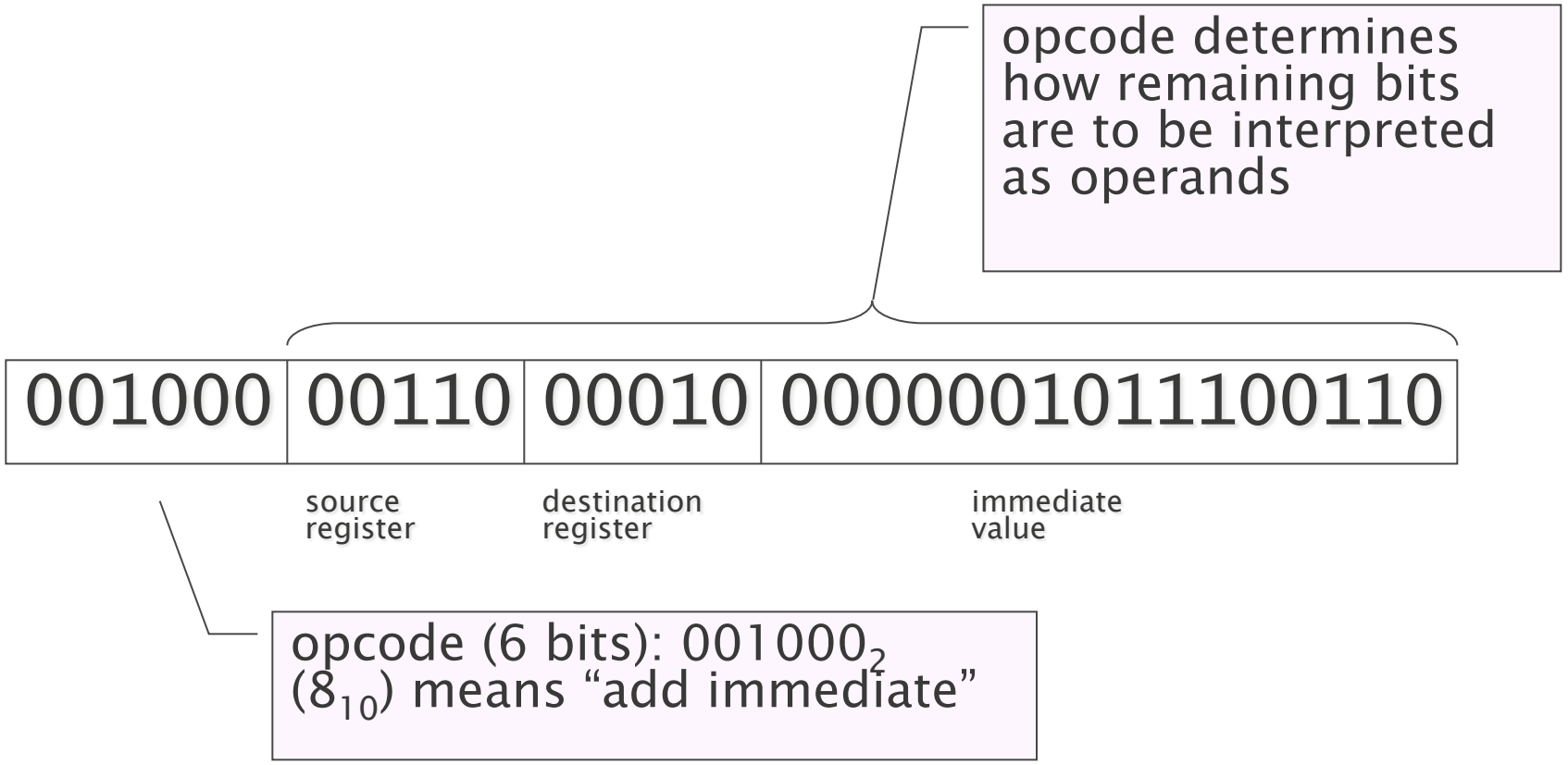
# "I-type" Instructions




- **All remaining instructions are I-type**
- **Immediate bit field used in three ways:**
  - In (real) load/store instructions
    - Actual address referenced is sum of reg_rs and imm field
  - In conditional branching instructions
    - imm is the "branch distance" measured in memory words from the address of the following instruction
  - In immediate ALU arithmetic or logic ops
    - Saves time where one operand is a small imm constant

# I-type Instruction – Example

- **Instruction's components encoded in binary**



# Going further

- **`if-elif-else` statement (`switch`)**
  - Efficiently selecting one of many options
  - Sometimes implemented with jump table (array of target addresses)
- **MIPS jump/branch delay slots**
  - Related to the instruction pipeline
  - Real MIPS CPU executes instruction after a branch or jump because it has already entered the instruction pipeline
  - Ignored in SPIM: configured for no delay slots
- **Do-whiles in Java**