Instructions and Conditional Execution

Andrea Janes

Content

- **Instruction Formats**
- R-format instructions
- I-format instructions
- Branches and I-format instructions
- J Operations
- Translation of typical loops
- A few more details

Instruction Formats

J-format: used for j and jal

I-format: used for instructions with immediates, lw and sw (since the offset counts as an immediate), and the branches (beq and bne), (but not the shift instructions; later)

R-format: used for all other instructions

- It will soon become clear why the instructions have been partitioned in this way.

Content

- *Instruction Formats*
- **R-format instructions**
- I-format instructions
- Branches and I-format instructions
- J Operations
- Translation of typical loops
- A few more details

R-Format Instructions (1/6)

- Define "fields" of the following number of bits each:

6	5	5	5	5	6

- For simplicity, each field has a name:

opcode	rs	rt	rd	shamt	funct

R-Format Instructions (2/6)

Important: Each field is viewed as a 5- or 6-bit unsigned integer, not as part of a 32-bit integer.

- *Consequence*: 5-bit fields can represent any number 0-31, while 6-bit fields can represent any number 0-63.

R-Format Instructions (3/6)

- What do these field integer values tell us?
  - **opcode**: partially specifies what instruction it is (Note: This number is equal to 0 for all R-Format instructions.)
  - **funct**: combined with opcode, this number exactly specifies the instruction
  - Question: Why aren't opcode and funct a single 12-bit field?
  - Answer: We'll answer this later.

R-Format Instructions (4/6)

More fields:

- **rs** (Source Register): *generally* used to specify register containing first operand
- **rt** (Target Register): *generally* used to specify register containing second operand (note that name is misleading)
- **rd** (Destination Register): *generally* used to specify register which will receive result of computation

R-Format Instructions (5/6)

- Notes about register fields:
  - Each register field is exactly 5 bits, which means that it can specify any unsigned integer in the range 0-31. Each of these fields specifies one of the 32 registers by number.
  - The word "generally" was used because there are exceptions

R-Format Instructions (6/6)

- Final field:
  - shamt: This field contains the amount a shift instruction will shift by. Shifting a 32-bit word by more than 31 is useless, so this field is only 5 bits (so it can represent the numbers 0-31).
  - This field is set to 0 in all but the shift instructions.
- For a detailed description of field usage for each instruction, see back cover of textbook.

R-Format Example - text

- MIPS Instruction:
  - add $8,$9,$10

opcode = 0 (look up in table)
funct = 32 (look up in table)
rs = 9 (first operand)
rt = 10 (second operand)
rd = 8 (destination)
shamt = 0 (not a shift)

R-Format Example - solution

decimal representation:

0	9	10	8	0	32

binary representation:

000000	01001	01010	01000	00000	100000

Called a Machine Language Instruction

Content

- *Instruction Formats*
- *R-format instructions*
- **I-format instructions**
- Branches and I-format instructions
- J Operations
- Translation of typical loops
- A few more details

I-Format Instructions (1/6)

- What about instructions with Immediates?
  - 5-bit field only represents numbers up to the value 31: immediates may be much larger than this
  - Ideally, MIPS would have only one instruction format (for simplicity): unfortunately, we need to compromise
- Define new instruction format that is partially consistent with R-format:

First notice that, if instruction has immediate, then it uses at most 2 registers.

I-Format Instructions (2/6)

- Define “fields” of the following number of bits each:

6	5	5	16

- Again, each field has a name:

opcode	rs	rt	immediate

Consistency (3/6)

- **Key Concept**: Only one field is inconsistent with R-format. Most importantly, `opcode` is still in same location.

I-Format Instructions (4/6)

- What do these fields mean?
  - **opcode**: same as before except that, since there's no funct field, opcode uniquely specifies an I-format instruction
  - This also answers question of why R-format has two 6-bit fields to identify instruction instead of a single 12-bit field: in order to be consistent with other formats.

I-Format Instructions (5/6)

- More fields:
  - **rs**: specifies the only register operand (if there is one)
  - **rt**: specifies register which will receive result of computation (this is why it is called the target register "rt")

I-Format Instructions (6/6)

Dealing with constants larger than 16 bits:

- lui: load upper immediate
  - **`lui $t2 253 #$t2 is register 10`**
- Loads in the 16 upper bits of register 10 the sequence 1111 1101
- So -for simplicity let1s assume 16 bits registers and 8 bits immediates, if I have to load 1111 1101 0010 1010 in $t2
  - **`Step 1: lui $t2 253`**
  - **`Step 2: addi $t2 $t2 42 #42 is 00101010`**

I-Format Example - text

MIPS Instruction:

- `addi $21,$22,-50`

opcode = 8 (look up in table)

rs = 22 (register containing operand)

rt = 21 (target register)

immediate = -50 (by default, this is decimal)

I-Format Example - solution

- MIPS Instruction:

```
addi    $21,$22,-50
```

decimal representation:

8	22	21	-50

binary representation:

001000	10110	10101	1111111111001110

Content

- *Instruction Formats*
- *R-format instructions*
- *I-format instructions*
- **Branches and J-format instructions**
- J Operations
- Translation of typical loops
- A few more details

How do we manage branches?

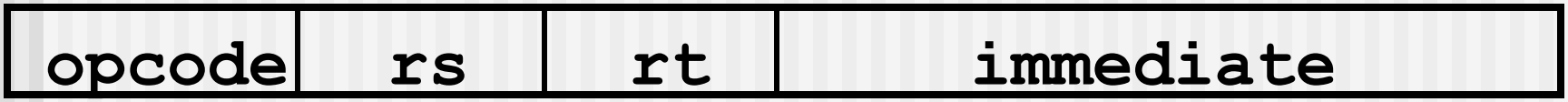
Branches correspond to several Java constructs:

- while(){…}
- do{…}while()
- for(…;…;…){…}
- switch(…){case x:…;break…}
- …

There are two key instructions: **beq** (branch if equal) and **bne** (branch if not equal)

Branches: PC-Relative Addressing (1/6)

- Use I-Format



- `opcode` specifies `beq` vs. `bne`
- `rs` and `rt` specify registers to compare

Branches: PC-Relative Addressing (2/6)

- What can `immediate` specify?
  - `Immediate` is only 16 bits
  - PC is 32-bit pointer to memory
  - So `immediate` cannot specify entire address to branch to.

Branches: PC-Relative Addressing (3/6)

- How do we usually use branches?
  - Answer: if-else, while, for
  - Loops are generally small: typically up to 50 instructions
  - Function calls and unconditional jumps are done using jump instructions (j and jal), not the branches.
- Conclusion: Though we may want to branch to anywhere in memory, a single branch will generally change the PC by a very small amount.

Branches: PC-Relative Addressing (4/6)

- Solution: PC-Relative Addressing
- Let the 16-bit immediate field be a signed two's complement integer to be added to the PC if we take the branch.
- Now we can branch +/- 2^{15} bytes from the PC, which should be enough to cover any loop.
- Any ideas to further optimize this?

Branches: PC-Relative Addressing (5/6)

- Note: Instructions are words, so they're word aligned (byte address is always a multiple of 4, which means it ends with 00 in binary).
  - So the number of bytes to add to the PC will always be a multiple of 4.
  - So specify the immediate in words.
- Now, we can branch +/- 2^{15} words from the PC (or +/- 2^{17} bytes), so we can handle loops 4 times as large.

Branches: PC-Relative Addressing (6/6)

- Final Calculation:
  - If we don't take the branch:

    PC = PC + 4
  - If we do take the branch:

    PC = **(PC + 4)** + (immediate * 4)
- Observations
  - Immediate field specifies the number of words to jump, which is simply the number of instructions to jump.
  - Immediate field can be positive or negative.
  - Due to hardware, add immediate to **(PC+4)**, not to PC; will be clearer why later in course

Branch Example (1/3)

- MIPS Code:

```
Loop:beq   $9,$0,End
     addi  $9,$9,-1
     j     Loop
End:
```

```
while (i!=0)
  i=i-1;
```

- Branch is I-Format:
  - `opcode` = 4 (look up in table)
  - `rs` = 9 (first operand)
  - `rt` = 0 (second operand)
  - `immediate` = ???

Branch Example (2/3)

- `Immediate` Field:
  - Number of instructions to add to (or subtract from) the PC, starting at the instruction *following* the branch.
  - In this case, `immediate` = 2

Branch Example (3/3)

- MIPS Code:

```
Loop:beq   $9,$0,End
     addi  $9,$9,-1
     j     Loop
End:
```

decimal representation:

4	9	0	2

binary representation:

000100	01001	00000	0000000000000010

Things to Remember

- Simplifying MIPS: Define instructions to be same size as data (one word): they can use the same memory (can use `lw` and `sw`).
- **Machine Language Instruction**: 32 bits representing a single instruction

R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	immediate		

- Computer actually stores programs as a series of these.

I-Format Problems (1/3)

- Problem 1:
  - Chances are that addi, lw, sw, … will use immediates small enough to fit in the immediate field.
- What if too big?
  - We need a way to deal with a 32-bit immediate in any I-format instruction.

I-Format Problems (2/3)

- Solution to Problem 1:
  - Handle it in software
  - Don’t change the current instructions: instead, add a new instruction to help out
- Use lui:

  lui register, immediate
  - takes 16-bit immediate and puts these bits in the upper half (high order half) of the specified register
  - sets lower half to 0s

I-Format Problems (3/3)

- So lui helps us…
  - Example:

```
addi   $t0,$t0, 0xABABCDCD
```

becomes:

```
lui    $at, 0xABAB
addi   $at, $at, 0xCDCD
add    $t0,$t0,$at
```

- Now each I-format instruction has only a 16-bit immediate.
- An instruction that must be broken up is called a *pseudoinstruction*. (Note that $at was used in this code.)

Content

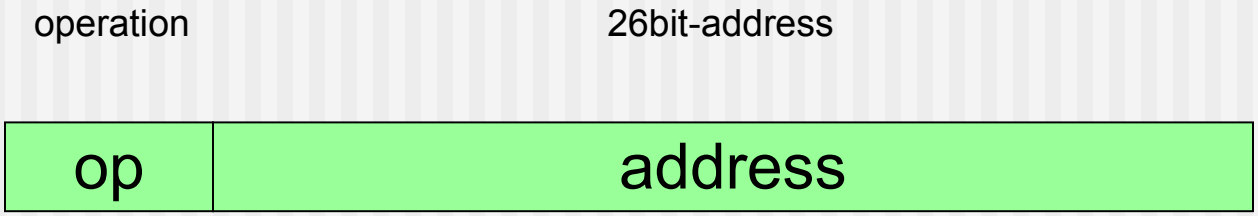
- *Instruction Formats*
- *R-format instructions*
- *I-format instructions*
- *Branches and I-format instructions*
- **J Operations**
- Translation of typical loops
- A few more details

Branch and jump

- **Two alternatives**
  - **Continue**
  - **Go to another program segment**
- **Branch after register comparison**
  - **Equal beq register1, register 2, L1**
  - **Not equal bne register1, register 2, L1**
  - **Register1 and register2 are compared**
  - **L1 is the target address when the condition is true**
- **Jump**
  - **Jump1: j address**
  - **Jump2: jr $s3 # address in reg. s3**

Instruction fields: J-type

- Used for specifying jump-address



Content

- *Instruction Formats*
- *R-format instructions*
- *I-format instructions*
- *Branches and I-format instructions*
- *J Operations*
- **Translation of typical loops**
- A few more details

Typical loops

- If
- If – then – else
- While
- Switch

If statement

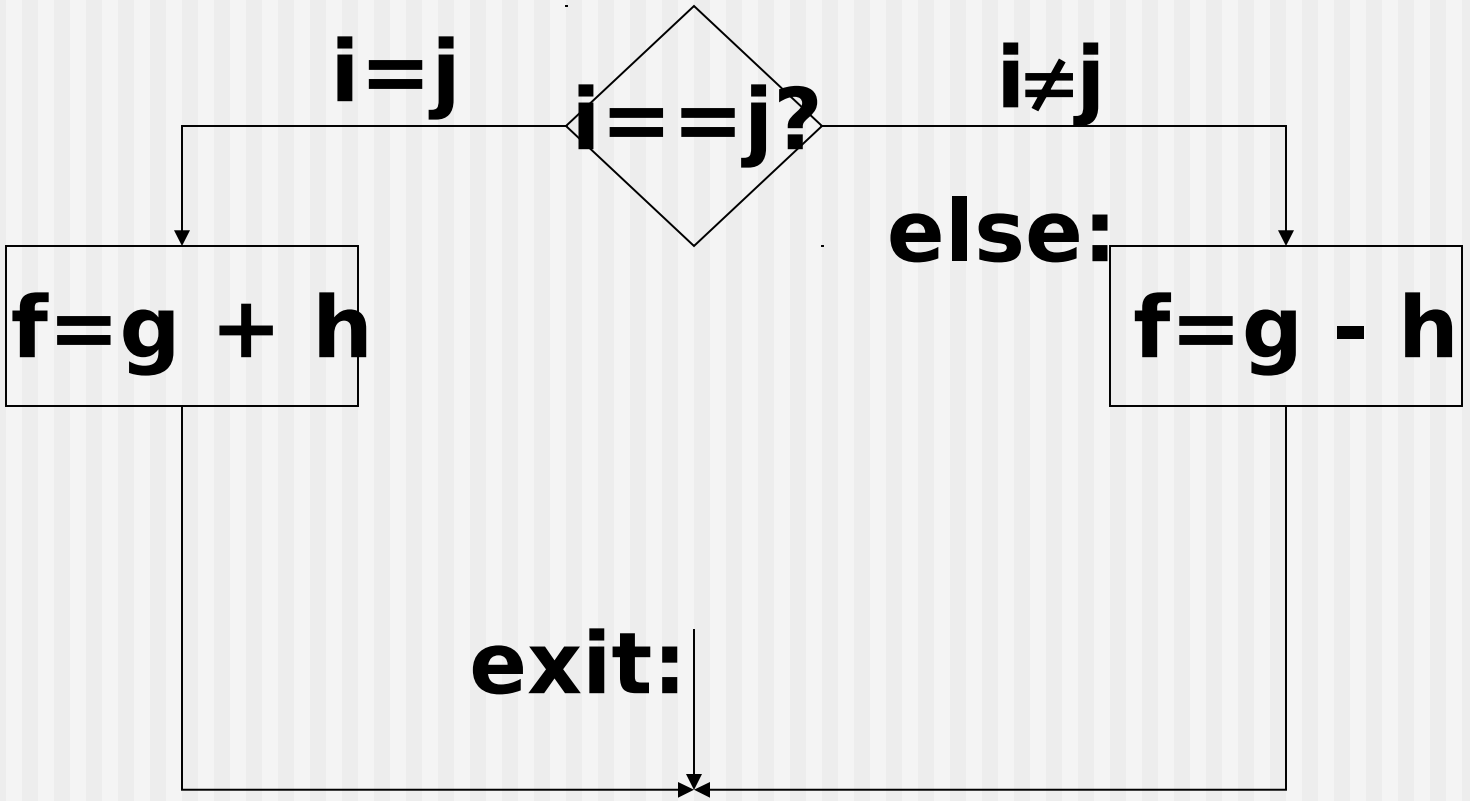
```
if (i != j) {
     f = g + h;
}
f = f - i;
```

Register assignment: $s0 ... $s4 = f ... j

```
     beq $s3, $s4, L1      #branch
     add $s0, $s1, $s2     #skipped
L1:  sub $s0, $s0, $s3     #always ex.
```

If – then – else statement

```
If (i == j) {
        f = g + h;
} else {
    f = g - h;
}
```



If – then – else statement

```
If (i == j) {
      f = g + h;
} else {
   f = g - h;
}
```

Registers $s0, $s1, $s2, $s3, $s4 = f, g, h, i, j

```
        bne  $s3, $s4, ELSE   #branch
        add  $s0, $s1, $s2    #skipped
        j    EXIT
ELSE:   sub  $s0, $s1, $s2
EXIT:   ...
```

While statement

```
while (save [i] == k) {
   i = i + j;
}
...
```

Registers: $s3, $s4, $s5 = i, j, k, array base = $s6

```
LOOP:       add   $t1, $s3, $s3     #$t1 = 2 • i
   add      $t1, $t1, $t1     #$t1 = 4 • i
   add      $t1, $t1, $s6     #$t1 = addr.
   lw  $t0, 0 ($t1)       #load
   bne      $t0, $s5, EXIT    #exit if !=
   add      $s3, $s3, $s4     #i = i + j
   j   LOOP
EXIT: ...
```

Switch Statement (1/3)

- Choose among four alternatives depending on whether k has the value 0, 1, 2 or 3. Compile this Java code:

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

Switch Statement (2/3)

- This is complicated, so **simplify**.
- Rewrite it as a chain of if-else statements, which we already know how to compile:

```
if(k==0) f=i+j;
   else if(k==1) f=g+h;
     else if(k==2) f=g-h;
       else if(k==3) f=i-j;
```

- Use this mapping:

f: $s0, g: $s1, h: $s2, i: $s3, j: $s4, k: $s5

Switch Statement (3/3)

```
     bne  $s5,$0,L1      # branch k!=0
     add  $s0,$s3,$s4    #k==0 so f=i+j
     j    Exit # end of case so Exit
L1:  addi $t0,$s5,-1          # $t0=k-1
     bne  $t0,$0,L2     # branch k!=1
     add  $s0,$s1,$s2     #k==1 so f=g+h
     j    Exit # end of case so Exit
L2:  addi $t0,$s5,-2          # $t0=k-2
     bne  $t0,$0,L3     # branch k!=2
     sub  $s0,$s1,$s2   #k==2 so f=g-h
     j    Exit # end of case so Exit
L3:  addi $t0,$s5,-3    # $t0=k-3
     bne  $t0,$0,Exit   # branch k!=3
     sub  $s0,$s3,$s4   #k==3 so f=i-j
Exit:
```


```
do j = j + 1
 while (_____);
```

Group exercise…

If $s3=i, $s4=j, $s5=@AWhat C code properly fills in the blank in the loop?

A: `A[i++] >= 10`
B: `A[i++] >= 10 |  A[i] < 0`
C: `A[i++] >= 10 &  A[i] < 0`
D: `A[i++] >= 10 || A[i] < 0`
E: `A[i++] >= 10 && A[i] < 0`
F: **None of the above**

Other decisions

- Set r1 on r2 less than r3
  - slt register 1, register 2, register 3
  - Compares two registers, reg. 2 and reg. 3
  - If the second is less than the third
    - register 1 = 1 else
    - register 1 = 0
- Example

```
slt     $8, $19, $20
```

- Branch less than

```
slt     $1, $10, $11          #$1 = 1 if $10 < $11
bne     $1, $0, LESS
```

To Summarize - Operands

MIPS operands		
Name	**Example**	**Comments**
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

To Summarize – Operations

MIPS assembly language				
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
Data transfer	load word	`lw $s1, 100($s2)`	`$s1 = Memory[$s2 + 100]`	Word from memory to register
	store word	`sw $s1, 100($s2)`	`Memory[$s2 + 100] = $s1`	Word from register to memory
	load upper	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return

Content

- *Instruction Formats*
- *R-format instructions*
- *I-format instructions*
- *Branches and I-format instructions*
- *J Operations*
- *Translation of typical loops*
- **A few more details**

A few more details

- More operations on immediates
- Another management of the upper-lower 16 bits
- More on Relative addressing

Immediate operands

- **Instructions with immediate operands**
  - **Add immediate: addi**
  - **Set less than immediate: slti**
  - **Or immediate: ori**
  - **And immediate: andi**
- **What happens with the upper 16 bits?**
  - **Lower 16 bits are loaded with immediate operand**
  - **addi: extends the leftmost of the 16 bits into the upper bits**
  - **ori: sets the upper 16 bits to 0**

Handling the upper 16 bits

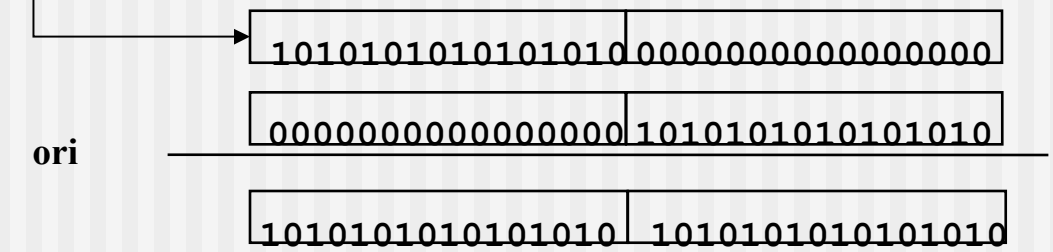
- We'd like to be able to load a 32 bit constant into a register
- Must use two instructions, new "load upper immediate" instruction

Loads upper and fills lower with zeros

```
lui $t0, 1010101010101010
```

Then must get the lower order bits right, i.e.,

```
ori $t0, $t0, 1010101010101010
```



Review of Relative addressing

- **Combination of a base register and the address in the branch operation**
- **PC = register + branch address**
- **Reference is the Program Counter, PC**
- **Relative jumps & branches**
- **No serious restriction**
- **High probability of the target being in the range of PC**

For the Missing Week ☺

Read the textbook

Review your notes

Do the exercises in the textbook

Read the textbook for next lecture:

www.unibz.it/informatic/courses/csa/schedule.htm