Lecture 10
Pseudo Instructions, Branching, & Control Structures in MIPS

```
## calculate A*X^2+B*X+C
## Output format must be:
## "answer = 180"
################################################
#                    text segment                    #
################################################

        .text
        .globl main
main:
        lw $t0,X
        lw $t1,A
        lw $t2,B
        lw $t3,C

        mul $t4,$t0,$t0 # t4 = X^2
        mul $t4,$t4,$t1 # t4 = A*X^2
        mul $t5,$t2,$t0 # t5 = B*X
        add $t4,$t4,$t5 # t4 = A*X^2+B*X
        add $t4,$t4,$t3 # t4 = A*X^2+B*X+C

        la $a0,ans          # system call to print
        li $v0,4  # out string
        syscall

        move $a0,$t4        # print result on terminal
        li $v0,1
        syscall

        la $a0,end1         # system call to print
        li $v0,4  # out a newline
        syscall

        li $v0,10
        syscall             # au revoir

################################################
#                    data segment                    #
################################################
        .data
X:      .word 7
A:      .word 3
B:      .word 4
C:      .word 5
ans: .asciiz "answer = "
end1:           .asciiz "\n"
```

This code is available to you on Sparky, in directory ~cse220/examples/spim

Load values of x, A, B, and C into temp registers

Register $t4 holds final value

Print the string “answer = ”

Print the result value

Print the newline

Exit the program

- *`mul`* is a pseudo instruction. Actual instruction is *`mult`*.
  - There is no actual corresponding machine language instruction.
  - The assembler will replace the `mul` instruction with one or more assembly instructions.
  - Why? For programming convenience.
  - <u>Ex:</u> In this program, we use `mul $t4, $t0, $t0` to calculate x^2, and store it in `$t4`.
    - Assembler does the following:
      ```
      mult $t0, $t0
      mflo $t4
      ```
    - Since `$t0` contains x, `mult` instruction will calculate x^2 in registers `Hi` and `Lo`.
    - Instruction `mflo` will copy lower 32 bits of product to `$t4`.
    - Thus `$t4` will have the product.
    - This requires that the product is less than 32 bits in length.

- Best way to study how pseudo instructions are assembled is to use them in a program and run it using SPIM.
  - Try:
    - `move $t1, $t2`
    - `add $t1, $zero, $t2`

- Several branch instructions are actually pseudo-instructions [conditional].
    - `la (load address)` is also a pseudo instruction.
    - `lw $t0,value` is also assembled slightly differently. Address of variable *value* is 32 bits long and we can not fit that in one instruction.

Branching

- Branch instructions allow for the changing of the flow of the program. Without them, the program will continue in a straight line of execution.

- Two types of branch instructions
    - Conditional branches
    - Unconditional/Direct branches

- *Conditional branches*

    - Branch which is dependent on a comparison. This comparison can be between 2 values in registers or with zero, and can be a signed or unsigned comparison.

    - Ex:
        - `beq` # branch on equal
        - `bgt` # branch on greater than
        - `beqz` # branch on equal to zero

    - Many conditional branches are pseudo-instructions. They are assembled using various 'set' instructions.

    - Ex:
        - `bge $t4, $t2, label` # branch to label if $t4 >= $t2

        - Assembled as:
          ```
          slt $1, $12, $10        # set register $1 to 1, if  register $12  <  register $10
          beq $1, $0, label       # branch to label if $1 = = 0
          ```

        - Register `$12` is `$t4`, register `$10` is `$t2`.
        - So if `$t4`< `$t2`, then set `$1` to 1, or `$1` is 0 if `$t4` >= `$t2`. This is what we want.
        - Next instruction branches if register `$1` is 0 .
        - This requires that register `$1` is available to the assembler. (`$1` is same as `$at`).

- *Unconditional branch*

    - Branch which always occurs.
    - Ex:
        - `b` # branch to label

- *Jump instructions*

    - Several types of jump instructions

        - `jr $ra` # jump to address in register
            - Specifically, this instruction is how you return from a function (more on functions later).
            - Can replace `$ra` with any register.

- `jal label`   # jump and link
    - This instruction is how you jump to a function.
    - Saves the return address in the `$ra` register (`$ra` = PC + 4), and then jumps to the label.
- `jalr Rsrc`   #jump and link register
    - This instruction jumps to the address stored in any register specified by `Rsrc` and places the return address in `$ra` (`$ra = ` PC + 4).

- Branch and jump instructions and their operand formats
    - Cross symbols represent pseudo-instructions.

Description	Opcode	Operands
Branch instruction	b†	label
Branch coprocessor z true	bczt	label
Branch coprocessor z false	bczf	label
Branch on equal	beq	Rsrc1, Src2, label
Branch on equal zero	beqz†	Rsrc, label
Branch on greater than equal	bge†	Rsrc1, Src2, label
Branch on GTE unsigned	bgeu†	Rsrc1, Src2, label
Branch on greater than equal zero	bgez	Rsrc, label
Branch on greater than equal zero and link	bgezal	Rsrc, label
Branch on greater than	bgt†	Rsrc1, Src2, label
Branch on greater than unsigned	bgtu†	Rsrc1, Src2, label
Branch on greater than zero	bgtz	Rsrc, label
Branch on less than equal	ble†	Rsrc1, Src2, label
Branch on LTE unsigned	bleu†	Rsrc1, Src2, label
Branch on less than equal zero	blez	Rsrc, label
Branch on greater than equal zero and link	bgezal	Rsrc, label
Branch on less than and link	bltzal	Rsrc, label
Branch on less than	blt†	Rsrc1, Src2, label
Branch on less than unsigned	bltu†	Rsrc1, Src2, label
Branch on less than zero	bltz	Rsrc, label
Branch on not equal	bne	Rsrc1, Src2, label
Branch on not equal zero	bnez†	Rsrc, label
Jump	j	label
Jump and link	jal	label
Jump and link register	jalr	Rsrc
Jump register	jr	Rsrc

Control Structure

- Any reasonable programming language must have an *if* statement, and some type of *loop*.

- Otherwise programs would be straight lines of code.

- Need conditional branch instructions and a direct or unconditional branch instruction.
- Ex:
    - `beq $t0, $t1, label`   # if $t0== $t1 then jump to the address of label
    - `beqz $t0, label`   # if $t0== 0 then jump to the address of label
    - `bne $t0, $t1, label`   # if $t0!= $t1 then jump to the address of label
    - `j label`   # jump to the address of label (direct branch)

- Code Structure for *if* and *while* statements
    - If <condition> then <then_part>, else <else_part>

	Code for testing or evaluating condition - Loading/calculating values - Loading/calculating addresses
	Branch statement to label **L1** (Ex: `bne $t0, $t1, L1`)
	Code for <else_part>
	`j` **`Loop_Done`**
L1:	Code for <then_part>
Loop_Done:	Code for <loop_done>

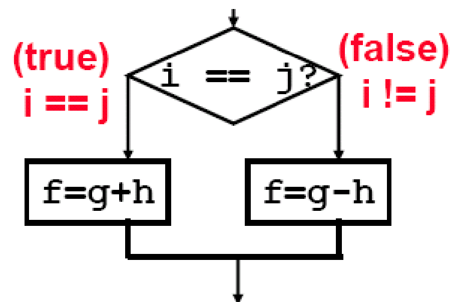
Implementation of an 'if'

°Compile by hand

C

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

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



MIPS

```
        beq $s3,s4, True   # branch i==j
        sub $s0,$s1,$s2    # f=g-h(false)
        j Exit             # go to Exit
True:   add $s0,$s1,$s2    # f=g+h (true)
Exit:
```

Implementation of a ‘do-while’

C

```
Loop:   g = g + A[i];
        i = i + j;
        if (i != h) goto Loop;
```

(g,h,i,j:$s1,$s2,$s3,$s4 : base of A[]:$s5)

MIPS

```
Loop: add $t1,$s3,$s3 #$t1= 2*i
      add $t1,$t1,$t1  #$t1= 4*i
      add $t1,$t1,$s5  #$t1=addr A
      lw  $t1,0($t1)   #$t1=A[i]
      add $s1,$s1,$t1  #g=g+A[i]
      add $s3,$s3,$s4  #i=i + j
      bne $s3,$s2,Loop# goto Loop
                      # if i!=h
```

Implementation of a ‘while’

C

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

(i,j,k: $s3,$s4,$s5: base of save[]:$s6)

MIPS

```
Loop: add $t1,$s3,$s3 #$t1 = 2*i
      add $t1,$t1,$t1 #$t1 = 4*i
      add $t1,$t1,$s6 #$t1=Addr
      lw  $t1,0($t1)  #$t1=save[i]
      bne $t1,$s5,Exit#goto Exit
                      #if save[i]!=k
      add $s3,$s3,$s4  # i = i + j
      j   Loop         # goto Loop

Exit:
```

Implementation of 'if less'

```
C  if (g < h) go to Less
--------------------------------------
M  slt $t0,$s0,$s1  # $t0 = 1 if
I                   # $s0<$s1 (g<h)
P  bne $t0,$zero, Less # goto Less
S                      # if $t0!=0
          . . .        # (if (g<h))
     Less:
```

A branch if $t0 != 0 branches if g < h.

- Register $zero always 0, so use bne comparing register $t0 to register $zero

° How test if (g >= h)?

Case/switch statement

° Choose among four alternatives depending on whether k has the value 0, 1, 2, or 3

```
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*/
}
```

- Notice that last case must wait for *n*-1 tests before executing, making it slow.
- Alternative tries to go to all cases equally fast: jump address table for scalability.
  - Idea: encode alternatives as a table of addresses of the cases.
    - Table is an array of words with addresses corresponding to case labels.
  - Program indexes into table and jumps.
- MIPS instruction "jump register" (`jr`) unconditionally branches to an address in a register; use load to get address.

```
    slti $t3,$s5,0      #Test if k < 0
    bne $t3,$zero,Exit  #if k<0,goto Exit
    slti $t3,$s5,4      # Test if k < 4
    beq $t3,$zero,Exit  #if k>=4,goto Exit
    add $t1,$s5,$s5     # Temp reg $t1 = 2*k
    add $t1,$t1,$t1     # Temp reg $t1 = 4*k
    add $t1,$t1,$t2    #$t1=addr JumpTable[k]
    lw  $t1,0($t1)     # $t1 = JumpTable[k]
    jr  $t1             # jump based on $t1
L0: add  $s0,$s3,$s4 # k=0 so f = i + j
    j    Exit    # end case, goto Exit
L1: add  $s0,$s1,$s2 # k=1 so f = g + h
    j    Exit    # end case, goto Exit
L2: sub  $s0,$s1,$s2 # k=2 so f = g - h
    j    Exit     # end case, goto Exit
L3: sub  $s0,$s3,$s4 # k=3 so f = i - j
Exit:       # end of switch statement
```