Lecture 3 & Lab 3 notes

More Conditional Operations

1. Instructions

```
# recap
beq rs, rt, L1 # if rs == rt, jump to label L1
bne rs, rt, L1 # if rs != rt, jump to label L1
# additionally
slt rd, rs, rt #if rs < rt, rd = 1
slti rs, rt, i #if rs < i, rs = 1
```

2. Compile condition operations

```
# suppose storing a in $s0, b in $s1
# if(a<b) L1; else L2
slt $t0, $s0, $s1
beq $t0, $zero, L2
L1: ...
j exit
L2: ...
exit:
...

# if(a<=b) L1; else L2
slt $t0, $s1, $s0 # b<a, t0 = 1
bne $t0, $zero, L2
L1: ...
j exit
L2: ...
exit:
...

# if(a>b) L1; else L2
slt $t0, $s1, $s0
beq $t0, $zero, L2
L1: ...
j exit
L2: ...
exit:
...

# if(a>=b) L1; else L2
slt $t0, $s0, $s1 # a<b, t0 = 1
bne $t0, $zero, L2 # t0 = 1,  j L2
L1: ...
j exit
L2: ...
exit:
```

```
...

# if(a==b) L1; else L2
bne $s0, $s1, L2
L1: ...
j exit
L2: ...
exit:
...

# if(a!=b) L1; else L2
beq $s0, $s1, L2
L1: ...
j exit
L2: ...
exit:
...
```

Loops

1. one layer loop

```
for(int i = 0; i < bound; i++){
    ...
}
```

```
li $s0, 0
li $s7, 8 # any bound you like
iLoop:
    addi $s0, $s0, 1
    ...
    beq $s0, $s7, exit
    j iLoop
exit:
    ...
```

2. 2 layers loop

```
for(int i = 0; i < 4; i++)
    for(int j = 0; j < i; j++){
        ...
    }
```

```
li $s0, 0
li $s1, 0
li $s2, 4
iLoop:
    addi $s0, $s0, 1
    ...
```

```
    jLoop:
        add $s1, $s1, 1
        ...
        bne $s1, $s0, exitj
        j jLoop
    exitj:
    li $s1, 0
    bne $s0, $s2, exiti
    j iLoop
exiti:
    ...
```

Procedure Calling

1. steps

   put **parameters** in a place where procedure can access them

   transfer control to the procedure

   acquire the storage resources needed for the procedures

   perform the tasks

   put the value in a place where the caller can access them

   return control to the caller

2. Implementation of step **1, 5, 6**

   4 argument-registers to pass the parameters: `$a0 -- $a3`

   2 return-registers to return the values: `$v0 -- $v1`

   store the origin that calls the function(**automatically**): `$ra`

   What if we need extra space(look at 4)?

   We can use other registers indeed, **but by conventions**, we use **memory** instead.

3. Implementation of step **2, 6** (More generally, how do we transfer control)

   caller use "jump and link" to jump to callee

   ```
   jal functionLabel # function label is self-defined
   ```

   callee use "jump and return to $ra" to jump to caller

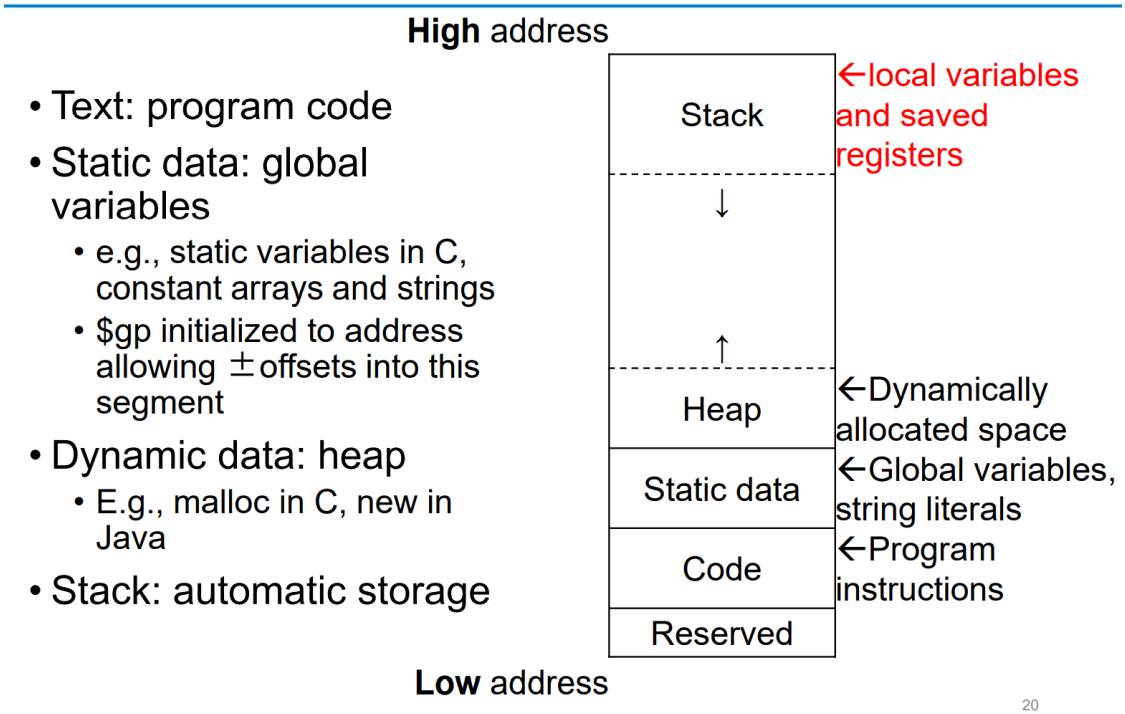
   ```
   jr $ra
   ```

   PC(program counter)

   PC is a special register maintain the address of the instruction being excuted. It will generally transfers to the next instruction except "j".

4. Stack pointer: providing extra register if needed

   1. Memmory alloction

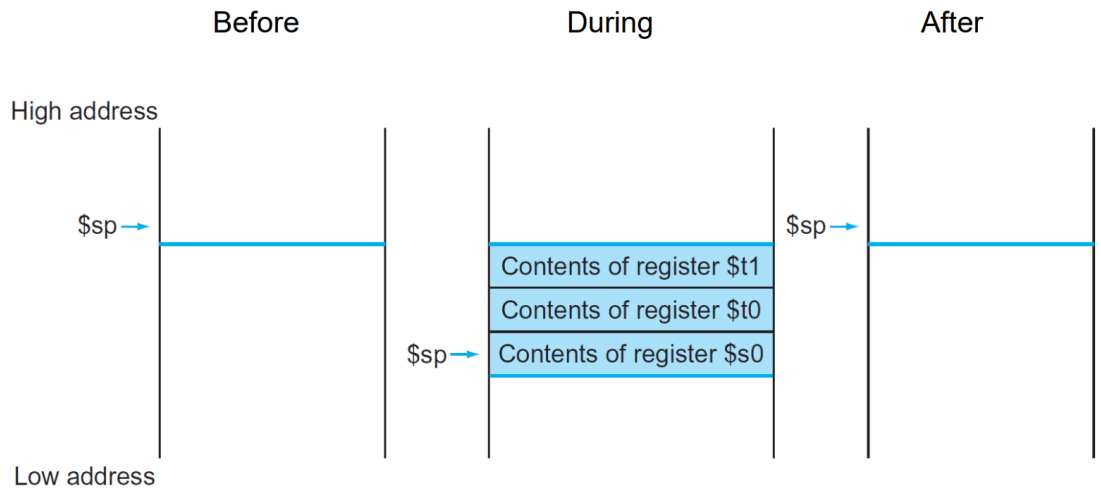


2. Idea of usage of stack pointer

   We want to use the registers like $s1, $s2 .... But as sourse register, they may be used in the caller, so what is a good solution?

   So we use stack to store the data of some registers used in the procedure, after which we recover the register with data in stack.

3. Stack before, during, after calling



5. a simple procedure example

```c
int leaf(int a, int b, int c, int d){
    return (a + b) - (c + d);
}
```

```
leaf:
    addi $sp, $sp, -4
    sw $s0, $sp

    add $s0, $a0, $a1
    sub $s0, $s0, $a2
    sub $s0, $s0, $a3
    addi $v0, $s0, 0

    lw $s0, $sp
    addi $sp, $sp, 4
    jr $ra
```

6. useful conventions to avoid memory operations
    1. caller's conventions:

       Can use VAT registers(such as $v0, $a0, $t0) freely

       Assume the parameter passed correctly

       Ensure return value to be stored in V registers

       make a copy for the used S registers

       Assume $ra is correct
    2. callee's conventions:

       Use S registers without fear of being overwritten

       Assume the return value is correct

       set $sp to the bottom of the stack(never point to some used data in the memory when calling a procedure)

       save $ra to the stack if neccessary(critial in compound function calls)
    3. summary of registers conventions

MIPS register conventions

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

Non-leaf procedure calling

1. an example:

```c
int fact(int n){
    if(n < 1)
        return 1;
    else return n * fact(n-1);
}
```

2. a feasible implementation

   Firstly, we should know that a recursive call looks like a stack. So, we can store data like a **stack**, i.e. **FIFO**

   Specifically, we can store the factor n in each step and the $ra each time until n equals to 0. Then we pack the stack pointer and manipulate the data in it.

3. codes

```assembly
store:
    addi $sp, $sp, -8
    sw $a0, $sp
    sw $ra, 4($sp)
    addi $a0, $a0, -1
    bne $a0, $zero, L1
    # when $a0 = 0
    # initialize the return value
    li $v0, 1
    jr $ra
L1:
    jal store # $ra is here or the main caller
Y:
```

```
    # pack the pointer,update the $ra
    addi $sp, $sp, 8
    lw $ra, 4($sp)
    # manipulate the data
    lw $t0, $sp
    mul $v0, $v0, $t0
    jr $ra
```

updating PC register

Knowing how PC register updates enable us to learn basic principle of program execution.

- Check if the current instruction is non-jump
  - If the current instruction is non-jump instruction: PC = PC+4
  - If the current instruction is jump instruction
    - If the current instruction is unconditional jump pc = destination address
    - If the current instruction is conditional jump
      - If the condition is met: PC = destination address
      - If the condition is not met: PC = PC + 4

Pseudo-instructions(A package of some instructions)

1. declaration:

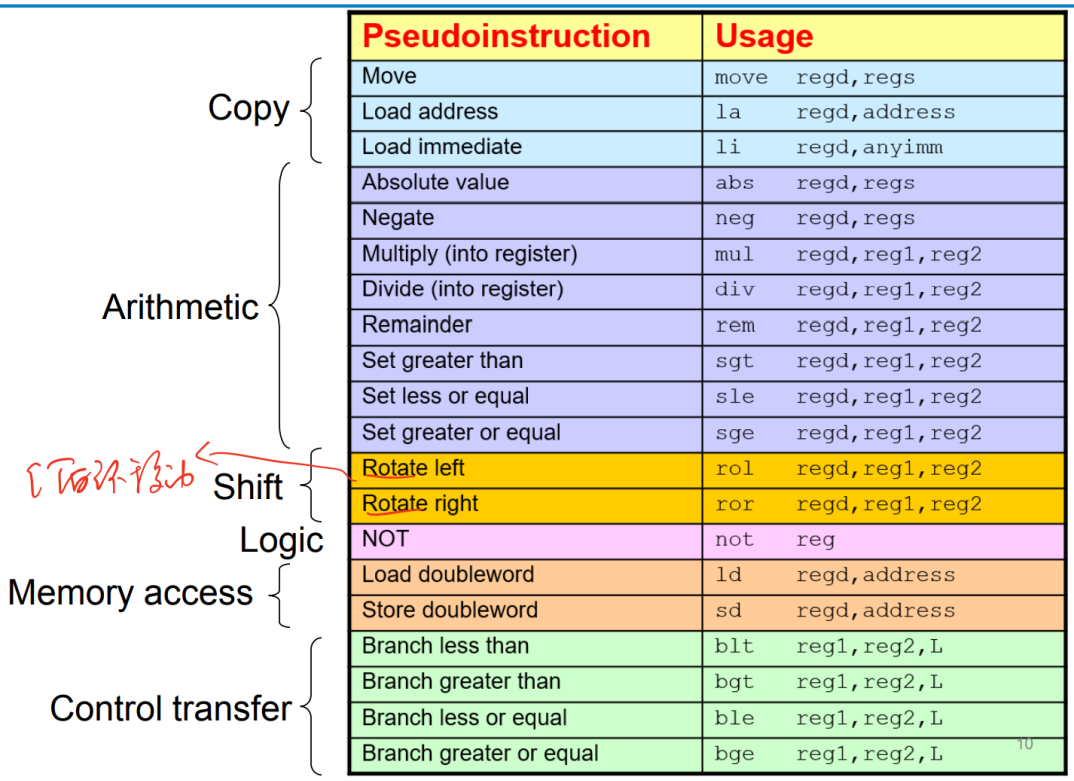
   there is no such instructions in the hardware.

   the assembler translates them into a combination of realinstructions

   we can use them to enhance our effieciency in the programming

2. examples

Pseudo-instructions



	Pseudoinstruction	Usage
Copy	Move	`move regd,regs`
Copy	Load address	`la regd,address`
Copy	Load immediate	`li regd,anyimm`
Arithmetic	Absolute value	`abs regd,regs`
Arithmetic	Negate	`neg regd,regs`
Arithmetic	Multiply (into register)	`mul regd,reg1,reg2`
Arithmetic	Divide (into register)	`div regd,reg1,reg2`
Arithmetic	Remainder	`rem regd,reg1,reg2`
Arithmetic	Set greater than	`sgt regd,reg1,reg2`
Arithmetic	Set less or equal	`sle regd,reg1,reg2`
Arithmetic	Set greater or equal	`sge regd,reg1,reg2`
Shift	Rotate left	`rol regd,reg1,reg2`
Shift	Rotate right	`ror regd,reg1,reg2`
Logic	NOT	`not reg`
Memory access	Load doubleword	`ld regd,address`
Memory access	Store doubleword	`sd regd,address`
Control transfer	Branch less than	`blt reg1,reg2,L`
Control transfer	Branch greater than	`bgt reg1,reg2,L`
Control transfer	Branch less or equal	`ble reg1,reg2,L`
Control transfer	Branch greater or equal	`bge reg1,reg2,L`

Summary of Useful Instructions

1. conditional jump

```
# basic instructions
bne $t0, $t1, L
beq $t0, $t1, L
# pseudo instructions
blt $t0, $t1, L # branch less than
ble $t0, $t1, L # branch less equal
bgt $t0, $t1, L # branch greater that
bge $t0, $t1, L # branch greater equal
bltu $t0, $t1, L # branch less than unsigned
bleu $t0, $t1, L # branch less equal unsigned
bgtu $t0, $t1, L # branch greater than unsigned
bgeu $t0, $t1, L # branch greater equal unsigned
```

2. loading

```
# we want to load data from a memory
lw $t0, address

# we want to load data from a register
lw $t0, ($t1) # parentheses cannot be omitted
addi $t0, $t1, 0
add $t0, $t1, $zero
```

```
move $t1, $t0

# we want to load a address to a register
la $t0, address
# after loading an address
4($t0) means the address of memory with offset 4, not $t1

# then we can use t0 as address, but parentheses cannot be omitted
lbu $t1, ($t0) # from the memory load a byte unsigned

# loading with different size
# follow the alignment, or throw address error
# n-byte alignment => address is a multiplication of n

lb $t0, 3($t1)  # no alignment request
lbu $t0, 3($t1)  # no alignment request

lh $t0, 2($t1)  # follow 2-byte alignment
lhu $t0, 2($t1)  # follow 2-byte alignment

lw $t0, 4($t1)  # follow 4-byte alignment

# ld means load 2 word, note that $t3 will be modified too.
ld $t0, 8($t3)  # follow 8-byte alignment

#loading without alignment
ulh $t0, 2($t1)
ulhu $t0, 2($t1)
ulw $t0, 4($t1)
uld $t0, 8($t3)
```

3. storing

```
# we only need to store register data to memory address
# register in mips follow big-endian, while in most computer system, memory follow little-endian.
#remember that storing is an operation of registers, i.e. la is needed to load an address.
la $t1, mem

# store a byte to memory
li $t0, 99
# automatically store the Least significant byte to (mem)
sb $t0, ($t1)

# a more instresting example
li $t0, 355 # 1*2^8 + 99
sh $t0, ($t1)
# (mem) will be 99 and 1(mem) will be 1
# Due to the difference of Big-endian and little-endian.
# instructions will automatically match the byte respectively
```

```
# we can conclude sw that ...

# storing with different size
# follow the alignment, or throw address error
# n-byte alignment => address is a multiplication of n

sb $t0, 3($t1)  # no alignment request

sh $t0, 2($t1)  # follow 2-byte alignment

sw $t0, 4($t1)  # follow 4-byte alignment

# sd means load 2 word, note that $t4 will be modified too.
sd $t0, ($t3)  # follow 8-byte alignment

# storing without alignment
ush $t0, 2($t1)
usw $t0, 4($t1)
usd $t0, 8($t3)
```

4. arithmetic instructions

```
# addition
add $t0, $t1, $t2
addi $t0, $t1, 5
addu $t0, $t1, $t2
addu $t0, $t1, 5

# subtraction
sub $t0, $t1, $t2
subu $t0, $t1, $t2
subu $t0, $t1, 5

# multiplication and division
# basic instructions
# store 64-bit, $t3 * $t4 = ($Hi, $Lo)
mult $t3, $t4
mfhi $t0 # $t0 = $Hi
mflo $t1 # $t1 = $Lo

# pseudo-instruction
mul $t0, $t3, $t4 # $t0 = $t3 * $t4

# multiplication
# basic instructions
# store 64-bit, $t3 * $t4 = ($Hi, $Lo)
mult $t3, $t4
mfhi $t0 # $t0 = $Hi, higher 32 bits
mflo $t1 # $t1 = $Lo, lower 32 bits

# pseudo-instruction(signed)
```

```
mul $t0, $t3, $t4 # ($t0, $t1) = $t3 * $t4

# for unsigned multiplication(only one method)
multu $t3, $t4
mfhi $t0 # $t0 = $Hi
mflo $t1 # $t1 = $Lo

# division
div $t3, $t4 # $Hi = $t3 % $t4, $Lo = $t3 / $t4
mfhi $t1 # $t1 = $Hi
mflo $t0 # $t0 = $Lo

# pseudo-instruction(signed)
div $t0, $t3, $t4 # $t0 = $t3 / $t4
rem $t1, $t3, $t4 # $t1 = $t3 % $t4

# for unsigned division(only one method)
divu $t3, $t4
mfhi $t1 # $t1 = $Hi
mflo $t0 # $t0 = $Lo

# set a "boolean"
# signed
slt $d, $s, $t
sgt $d, $s, $t
sle $d, $s, $t
sge $d, $s, $t
# unsigned
sltu $d, $s, $t
sgtu $d, $s, $t
sleu $d, $s, $t
sgeu $d, $s, $t
```

5. shifting

```
# $shamt is in bits
sll $d, $t, shamt
sla $d, $t, shamt
sra $d, $t, shamt
# rotate left, complements left overflowed bits to the right
rol $d, $t, shamt
# rotate right, complements right overflowed bits to the left
ror $d, $t, shamt
```

6. logic

```
and $d, $s, $t
or $d, $s, $t
xor $d, $s, $t
not $d, $s # $d = ~$s
nor $d, $s, $t # $d = ^($s | $t)
```