MIPS memory map

Static data is known at the time the program is loaded into memory.

Dynamic data is allocated at run time.
Fetch, Decode, Execute

Instruction cycle:

1. Fetch the next instruction from memory
2. Decode it
3. Execute it
4. Update the PC (program counter) += 4
Register PC - Program Counter

Register PC is updated by 4 after every instruction.

You can see this when you single-step through a program.

In the code below, we are about to execute 0x004000c and we see that is the value in register PC.
Control structures

So far we have executed code sequentially.

We need:

- Conditional execution, like an if statement
- Repeated execution, like loops
- Function calls

In assembly language we use branch and jump instructions to create these
Branch instructions

Branch instructions are conditional jumps

Branch to a labeled instruction if a condition is true; otherwise continue sequentially

There are two MIPS branch instructions: beq and bne

beq rs, rt, label  # compare registers and branch if they are equal

bne rs, rt, label # compare registers and branch if they are not equal
Jump

The jump instruction is unconditional.

j label    # start executing the code at label

This will cause the current value of the PC to be replaced by the address “label”
Implementing an if statement

C code:

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

f, g, ... in $s0, $s1, ...

Compiled MIPS code:

```mips
bne $s3, $s4, Else
add $s0, $s1, $s2
j Exit
```

Else:

```mips
sub $s0, $s1, $s2
```

Exit:

Assembler calculates addresses
IF-ELSE example

We jump over the add if the condition is false.

We have to jump over the sub if the condition is true.
### Assembled Code

<table>
<thead>
<tr>
<th>Bkpt</th>
<th>Address</th>
<th>Code</th>
<th>Basic</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x00000000c</td>
<td>0x8c510004</td>
<td>lw $1, 0x00000004(s1)</td>
<td></td>
</tr>
<tr>
<td>0x00400010</td>
<td>0x3c011001</td>
<td>lui $1, 0x000000101</td>
<td>14: lw $s2, h</td>
<td></td>
</tr>
<tr>
<td>0x00400014</td>
<td>0x8c320008</td>
<td>lw $18, 0x00000008(s1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00400018</td>
<td>0x3c011001</td>
<td>lui $1, 0x000001001</td>
<td>15: lw $s3, i</td>
<td></td>
</tr>
<tr>
<td>0x0040001c</td>
<td>0x8c33000c</td>
<td>lw $19, 0x0000000c(s1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00400020</td>
<td>0x3c011001</td>
<td>lui $1, 0x00000001</td>
<td>16: lw $s4, j</td>
<td></td>
</tr>
<tr>
<td>0x00400024</td>
<td>0x8c340010</td>
<td>lw $20, 0x00000010(s1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00400028</td>
<td>0x16740002</td>
<td>bne $19, $20, 0x00000002</td>
<td>18: bne $s3, $s4, Else</td>
<td></td>
</tr>
<tr>
<td>0x0040002c</td>
<td>0x12320000</td>
<td>add $16, $17, $18</td>
<td>19: add $s0, $s1, $s2</td>
<td></td>
</tr>
<tr>
<td>0x00400030</td>
<td>0x0810000e</td>
<td>j 0x00400038</td>
<td>20: j Exit</td>
<td></td>
</tr>
<tr>
<td>0x00400034</td>
<td>0x12320002</td>
<td>sub $16, $17, $18</td>
<td>21: Else: sub $s0, $s1, $s2</td>
<td></td>
</tr>
<tr>
<td>0x00400038</td>
<td>0x2402000a</td>
<td>addiu $2, $0, 0x00000000a</td>
<td>23: Exit: li $v0, 10</td>
<td></td>
</tr>
<tr>
<td>0x0040003c</td>
<td>0x00000000c</td>
<td>syscall</td>
<td>24: syscall</td>
<td></td>
</tr>
</tbody>
</table>
Branch statements use the I format

beq rs, rt, label

bne rs, rt, label

The offset is relative to the current PC value.
Branch v. Jump

j label

The jump instruction is absolute, the PC is updated to point to label. We can jump anywhere in the code segment.

beq $t1, $t2, label

The branch instruction is relative to the current value of the PC. The 16-bit offset is added to the PC. If the offset is positive, it's a forward jump; if the offset is negative, it's a backward jump
branch addressing

Most branch targets are close, so a 16-bit offset is sufficient.

At a branch instruction, the PC is already pointing to the next instruction (PC+4)

target address = PC + offset*4

the offset is in words (4 bytes)

If the branch target is too far away, the assembler will rewrite it with jump.
MIPS has 3 instruction formats

<table>
<thead>
<tr>
<th>BASIC INSTRUCTION FORMATS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R</strong></td>
</tr>
<tr>
<td>31</td>
</tr>
<tr>
<td><strong>I</strong></td>
</tr>
<tr>
<td>31</td>
</tr>
<tr>
<td><strong>J</strong></td>
</tr>
<tr>
<td>31</td>
</tr>
</tbody>
</table>
Jump Decoding

The J instruction format has 6 bits for the opcode, leaving 26 bits for the label. But addresses are 32 bits, so how does that work? At run time:

- First the 26 bit address is shifted left twice to become 28 bits
- Then the upper 4 bits of the PC are appended to the MSB of the address

Ex: 0x0810000c -> 000010 (opcode 2 hex) and: 000000000000000000000000011000

Shift left twice: 0000000000000000000000110000

Append 4 MSB of PC: 0000000000000000000000000110000B
Loops: counter loop

```assembly
.data
n: .word 3
count: .word 0

.text
li $t1, 0  # $t1 = i = 0
lw $t2, n  # $t2 = stop value
loop:
  beq $t1, $t2, done  # branch if i == 3
  addi $t1, $t1, 1  # i++
  j loop

done:
  sw $t1, count  # save i

exit:
  li $v0, 10
  syscall
```
Loops: looping through an array

```assembly
# looping through an array
# while (arr[i] != -1) i++;

.data
arr: .word 3, 8, 12, -1

.text
li    $s3, 0        # $s3 = i = 0
la    $s6, arr      # $s6 = base address of array
li    $s5, -1       # $t9 = k

loop:  sll  $t1, $s3, 2    # i = i * 4
add   $t1, $t1, $s6   # address = i*4 + arr[0]
lw    $t0, ($t1)      # get next array element
beq   $t0, $s5, exit # if arr[i] == -1, exit
addi  $s3, $s3, 1    # i++
j     loop           # goto next iteration

exit:  li    $v0, 10
syscall
```
Conditional statements

The beq and bne instructions can be used to create relational conditions like >, <=

First a condition is checked with slt (set less than) or slti instruction. Instruction slt or slti will set Rd to 1 if the condition is true, 0 otherwise.

Then a branch is taken, or not, based on if the condition is equal to $zero.

# slt example
slt $t0, $s3, $s4  # $t0 = $s3<$s4
beq $t0, $zero, label
# will branch if NOT $s3<$s4

# slti example
slti $t0, $s3, 10  # $t0 = $s3<10
bne $t0, $zero, label
# will branch if $s3<10
slt and slti

slt rd, rs, rt  # set rd=1 if rs<rt; otherwise rd=0

slti rd, rs, constant  # set rd=1 if rs<constant; otherwise rd=0

Used immediately before beq or bne.

Can be used to implement any conditional (<, <=, >, >=) by changing the order of the source operands

Why no blt, etc?

Two fast instructions are better than one slower one.
signed v. unsigned

signed comparison: slt and slti

unsigned comparison: sltu and sltiu
array bounds check

An unsigned comparison checks if $x < y$ and also if $x$ is negative.

Case 1: $s1>s2$ indicates we have gone beyond the end of the array.

Case 2: $s1$ is negative.

$s1$ will be ">" $t2$ with an unsigned check because it will have 1 in MSB.

# jump to IndexOutOfBounds
#   if $s1>$t2 or $s1$ is negative
sltu $t0, $s1, $t2
beq $t0, $zero, IndexOutOfBounds
Pseudo-instructions for branches

These get converted into slt-beq or slt-bne instructions.

blt - branch less than

ble - branch less than or equal to

bgt - branch greater than

bge - branch greater than or equal to

blt $t1, $t2, exit
# will be assembled into:
slt $1, $9, $10
bne $1, $0, exit

ble $t1, $t2, exit
# will be assembled int:
slt $1, $10, $9
beq $1, $0, exit
if example

MIPS:

```assembly
# $s0 = f, $s1 = g, $s2 = h, $s3 = i, $s4 = j
bne $s3, $s4, L1  # if i != j, skip if block
add $s0, $s1, $s2  # if block: f = g + h
L1:
sub $s0, $s0, $s3  # f = f - i
```

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

MIPS code:

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

```mips
# $s0 = f, $s1 = g, $s2 = h, $s3 = i, $s4 = j
bne $s3, $s4, else  # if i ! = j, branch to else
add $s0, $s1, $s2  # if block: f = g + h
    j L2            # skip past the else block
else:
    sub $s0, $s0, $s3  # else block: f = f - i
L2:
```
while loop

```c
int pow = 1;
int x = 0;

while (pow != 128)
{
    pow = pow * 2;
    x = x + 1;
}
```

```
# $s0 = pow, $s1 = x
addi $s0, $0, 1    # pow = 1
addi $s1, $0, 0    # x = 0
addi $t0, $0, 128  # t0 = 128 for comparison
while:
    beq $s0, $t0, done  # if pow == 128, exit while
    sll $s0, $s0, 1    # pow = pow * 2
    addi $s1, $s1, 1   # x = x + 1
    j while
done:
```
for loop

MIPS code:

```
int sum = 0;

for (i = 0; i != 10; i = i + 1) {
    sum = sum + i;
}
```

// equivalent to the following while loop
int i = 0;
while (i != 10) {
    sum = sum + i;
    i = i + 1;
}

```c

    # $s0 = i, $s1 = sum
    add $s1, $0, $0    # sum = 0
    addi $s0, $0, 0   # i = 0
    addi $t0, $0, 10  # $t0 = 10
    for:
        beq $s0, $t0, done  # if i == 10, branch to done
        add $s1, $s1, $s0  # sum = sum + i
        addi $s0, $s0, 1   # increment i
    j for

done:
```
functions aka procedures aka subroutines

Steps to calling a function:

In calling code:

1. place arguments in registers
2. transfer control to procedure
3. process any return values

In the called procedure:

1. acquire storage (stack) for procedure if needed
2. perform procedure's operations
3. place results in register for caller
4. return to place of call
MIPS registers for functions

$\text{a0 - a3} - \text{arguments for the function}

$\text{v0, v1} - \text{return values from the function}

$\text{t0 - t9} - \text{temporaries (may be overwritten by the function)}

$\text{s0 - s7} - \text{saved (function must save/restore them on the stack)}

$\text{sp} - \text{stack pointer, points to the top of the stack}

\text{Not important in MARS: fp frame pointer, gp global pointer}
How to call functions in MIPS

Call a function:

jal ProcedureLabel

jal "jump and link:
- first saves $pc to $ra so we can get back
- then jumps to ProcedureLabel

As we execute jal, $pc will already be pointing to the instruction immediately after jal; we need to save this return address in $ra

Return from a function:

jr $ra

jr "jump register" will jump to the value in $ra
it copies the $ra to the $pc so that the next instruction to be executed is after the jal
leaf function

```assembly
# leaf function
# result = sum(x, y)

.data
x: .word 3
y: .word 5
result: .word 0

.text
lw $a0, x
lw $a1, y
jal sum
sw $v0, result

exit: li $v0, 10
syscall

sum:  # return x + y
      # x and y are in $a0 and $a1
      # sum is returned in $v0
add $v0, $a0, $a1
jr $ra
```
Problem 1

```
 1  # practice program 1
 2  # if (a < 0) a = -a
 3     .data
 4     a:    .word    4
 5
 6     .text
 7     main:
 8     # your code here
 9
10
11
12
13     exit:  li  $v0, 10
14     syscall
15```
Problem 2

```assembly
# practice program 2
# if (a > 0) a = -a
.data
a:  .word 4  # change to negative to test

.text
main:
# your code here

exit:  li $v0, 10
       syscall
```
Problem 3

```assembly
# practice program 3
# if (a <= b) c = b else c = a
.data
da: .word 5
db: .word 6
dc: .word 0
text
main:
exit: li $v0, 10
syscall
```
Problem 4

```assembly
# practice program 4
# for (i=0; i<10; i++) c += 5;  # use immediate load/add instructions

.data
    c: .word 0

.text
main:

exit:   li $v0, 10
        syscall
```
Problem 5

```assembly
# practice program 5
# for (i=0; i<10; i++) a[i] +=5;
.data
a: .word 5, 9, 2, 1, 4, 6, 3, 9, 2, 1
len: .word 10
.text
main:
exit: li $v0, 10
syscall
```
Problem 6

```assembly
  # practice program 6
  # while (s2[i] = s1[i] != '/0') i++;
  .data
  s1: .asciiz "hi"
  .align 2
  s2: .space 4
  .text
  main:

  exit:   li $v0, 10
          syscall
```

Problem 7

move the loop in Problem 7 to a subroutine that you call from the main program