Tutorial 3

- Mips Loops
- Mips Arrays
- Symbol Tables

Mips Loops

- Done via branching
- Assembly:

```
beq $s, $t, i ; If $s == $t then branch with offset i
bne $s, $t, i ; If $s != $t then branch with offset i
```

↳ i can be +ve to move PC forward of -ve to move PC backwards

- Machine Code

```
beq : 000100 sssss ttttt iiii iiii iiii iiii
bne : 000101 sssss ttttt iiii iiii iiii iiii
```

↳ i is encoded in 16-bit 2's compliment

- How does branching with offset i work?
- Recall the fetch-execute cycle:

```
PC = 0
while PC != $31 :
        IR = MEM[PC]      // get instruction at MEM[PC]
        PC = PC + 4       // next 4 byte (32-bit) instruction
        ... run IR's instruction...
done
```

- beq & bne will modify PC by adding i
  - ↳ Implicit conversion : $PC = PC + 4i$
- Loop idea :

n−1 words { ... loop start ... ⟵ Keep running the loop until bne condition fails
n−2 lines
bne \$_, \$_, −n
... next line ... ⟵ when bne runs, PC is pointing to the next instruction

- Annoying to hard-code loop offset i, easier to use labels

start: ... loop start ... ⟵ Same loop!
n−2 lines
bne \$_, \$_, start
... next line ...

eg: Write a MIPS program that takes non-negative integer n in $1 & stores the factorial n! into $3

soln)

```
; Initialize the answer $3 = 1 & $11 = 1
lis $3
.word 1
add $11, $3, $0

; Loop until $1 = 0
loop:  beq $1, $0, end
       mult $3, $1
       mflo $3              ; $3 = $3 * $1
       sub $1, $1, $11      ; $1 = $1 - 1
       beq $0, $0, loop
end:   jr $31
```

eg: Recall the Fibonacci sequence def^n:

$$f_0 = 0$$
$$f_1 = 1$$
$$f_{n+2} = f_{n+1} + f_n \quad \text{for } n \geq 0$$

Write a MIPS program which takes non-negative integer n in \$1 & stores f_n in \$3

sol^n)

```
add $3, $0, $0     ; $3 = f0
lis $4
.word 1            ; $4 = f1
add $11, $4, $0    ; $11 = 1

; Loop until $1 = 0
loop :  beq $1, $0, end
        add $5, $4, $0    ; $5 = f_{i+1}
        add $4, $3, $4    ; $4 = f_{i+2} = f_{i+1} + f_i
        add $3, $5, $0    ; $3 = $5 = f_{i+1}

        sub $1, $1, $11   ; $1 = $1 - 1
        beq $0, $0, loop
end :   jr $31
```

MIPS Arrays

- We can use mips.array to write programs that manipulate arrays!
  - ↳ lets us write programs that can accept > 2 inputs!

eg Write a MIPS program that accepts the address of an array in $1 & its length in $2 & stores the product of the numbers in the array in $3.

soln)

```
add $2, $2, $2
add $2, $2, $2
add $2, $2, $1    ; $2 = 4*$2 + $1 (last array address)
lis $4
.word 4
lis $3
.word 1

; Loop until $1=$2, incrementing $1 by 4 each loop
loop:  beq $1, $2, end
       lw $5, 0($1)      ; $5 = *($1) = Arr[i]
       mult $3, $5
       mflo $3           ; $3= $3 * Arr[i]
       add $1, $1, $4    ; $1= $1+4 (i=i+1, next Arr index)
       beq $0, $0, loop
end:   jr $31
```

Symbol Tables

- Assembler divided into 2 phases
  1) Analysis
     ↳ Checks input & instruction correctness
     ↳ Construct a Symbol Table
        → Stores the values of all labels defined in code
  2) Synthesis
     ↳ Uses symbol table to substitute labels with their values & compute branch (bne/beq) offsets (i)
- ☆ labels are why we do 2 passes over the code
  ↳ Can't combine Analysis & Synthesis since we can't know if a label is being used before it is defined!
- During the second pass:
  ↳ Check for uses of undefined labels
  ↳ Check that label operands, when converted to addresses or offsets, fall in the correct ranges

  These require a complete symbol table! Hence the 2nd pass
- ☆ Remember to check for duplicate label defⁿs when constructing a symbol table!

eg: Construct the symbol table for the following code:

```
begin:
label: beq $0, $0, after
jr $4

after:
sw $31, 16($0)
lis $4
abc0: abc1: .word after

loadStore:
lw $20, 4($0)
sw $20, 28($0)

end:
```

Sol^n:

↳ label values are the number of non-null (lines with instructions) that preceed the label multiplied by 4.

```
               begin:
7 5 4 4 2      label: beq $0, $0, after
6 4 3 3 1      jr $4
               ; Null line, not counted
               after:
5 3 2 2        sw $31, 16($0)
4 2 1 1        lis $4
3 1            abc0: abc1: .word after
               ; Null line, not counted
               loadStore:
2              lw $20, 4($0)
1              sw $20, 28($0)
               ; Null line, not counted
               end:
```

Symbol Table:

begin :	0	(no instructions preceed begin:)
label :	0	("beq" does not preceed label: def^n)
after :	8	(2 preceeding instructions)
abc0 :	16	(.word comes after both labels,
abc1 :	16	4 instruction lines)
loadStore :	20	(5 preceeding instruction lines)
end :	28	(8 " " "