Tutorial 3

- Mips Loops
- Mips Arrays
- Symbol Tables

Mips Loops

- Done via branching

Assembly:
\[
\begin{align*}
&\text{beg: } \text{if } s_5 = s_6 \text{ then branch with offset } i \\
&\text{bne: } \text{if } s_5 \neq s_6 \text{ then branch with offset } i
\end{align*}
\]

\( \Rightarrow \) i can be tve to move PC forward or -ve to move PC backwards

Machine Code:
\[
\begin{align*}
&\text{beg: } 000100 \text{ ss ss ss s t t t t i i i i i i i i} \\
&\text{bne: } 000101 \text{ ss ss ss s t t t t i i i i i i i i}
\end{align*}
\]

\( \Rightarrow \) i is encoded in 16-bit 2's complement

- How does branching with offset i work?

- Recall the fetch-execute cycle:

\[
\begin{align*}
&\text{PC = 0} \\
&\text{while } \text{PC } \neq \$31 \\
&\quad \text{IR = MEM[PC]} \quad \// \text{get instruction at MEM[PC]} \\
&\quad \text{PC = PC + 4} \quad \// \text{next 4 byte (32-bit) instruction} \\
&\quad \text{... run IR's instruction...} \\
&\text{done}
\end{align*}
\]
• beg & bne will modify PC by adding i
  \[ \text{PC} = \text{PC} + 4i \]
  \[ \Rightarrow \text{Implicit conversion: } \text{PC} = \text{PC} + 4i \]

• Loop idea:

\[
\begin{align*}
\text{(n-1) words} & \quad \text{keep running the loop until} \\
\text{n-2 lines} & \quad \text{bne condition fails} \\
\text{bne } & \quad \text{next line} \quad \text{when bne runs, PC is pointing to the next instruction}
\end{align*}
\]

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

\[
\begin{align*}
\text{start: } & \quad \text{loop start} \quad \text{Same loop!} \\
\text{n-2 lines} & \quad \text{bne } \_\_\_, \_\_\_, \text{start} \\
\text{next line} & \quad \text{next line}
\end{align*}
\]
eg: Write a MIPS program that takes non-negative integer n in $t$ & stores the factorial $n!$ into $3$ so

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

; Loop until $1 = 0$
loop: beg $1, $0, end
mult $3, $1
mflo $3 ; $3 = $3 * $1
sub $1, $1, $11 ; $1 = $1 - 1
beg $0, $0, loop
end: jr $31
Recall the Fibonacci sequence defined:

\[
\begin{align*}
    f_0 &= 0 \\
    f_1 &= 1 \\
    f_{n+2} &= f_{n+1} + f_n & \text{for } n \geq 0
\end{align*}
\]

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

```
add $3, $0, $0 ; $3 = f_0
lis $4
.word 1 ; $4 = f_1
add $11, $4, $0 ; $11 = 1

i 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!

  Let's us write programs that can accept > 2 inputs!

  Example: 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.

```mips
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 = *(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 def’s when constructing a symbol table!
eg: Construct the symbol table for the following code:

```
begin:
label: beg $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:
```
So

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

Symbol Table:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>0</td>
<td>(no instructions precede begin:)</td>
</tr>
<tr>
<td>label</td>
<td>0</td>
<td>(beg “does not precede label: def”)</td>
</tr>
<tr>
<td>after</td>
<td>8</td>
<td>(2 preceding instructions)</td>
</tr>
<tr>
<td>abc0</td>
<td>16</td>
<td>(.word comes after both labels:)</td>
</tr>
<tr>
<td>abc1</td>
<td>16</td>
<td>(4 instruction lines)</td>
</tr>
<tr>
<td>load Store</td>
<td>20</td>
<td>(5 preceding instruction lines)</td>
</tr>
<tr>
<td>end</td>
<td>28</td>
<td>(8 “ ” “ ” )</td>
</tr>
</tbody>
</table>