Tutorial 04

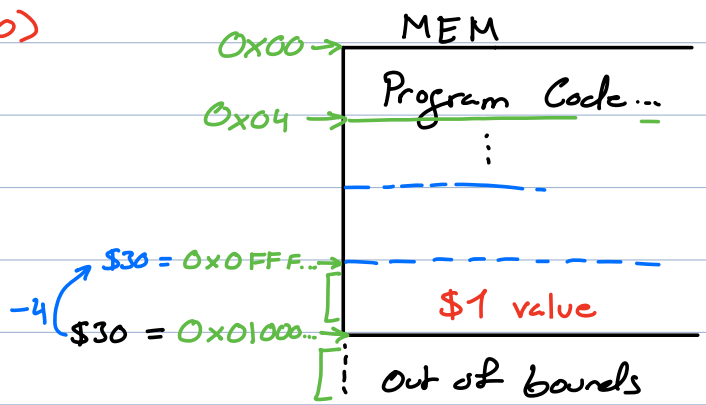
- Stacks
- Procedures

Stacks

- $30 = Stack Pointer
  - ↳ initially an out of bounds address (empty)
  - ↳ grows backwards every 4 bytes
- Pushing to the Stack
  1) Store the word 4 bytes before $30 (i = -4)
  2) set $30 = $30 - 4 (points at the top of the stack)

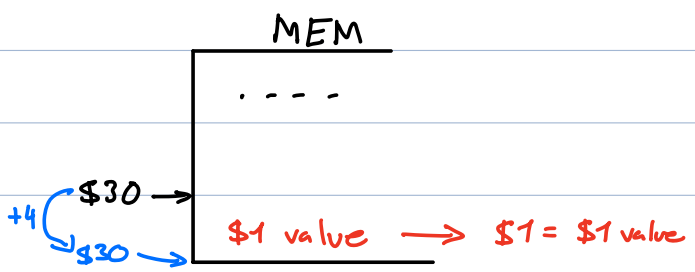
ie: Push register $1 onto the stack

```
sw   $1  -4 ($30)
lis  $1
.word 4
sub  $30, $30, $1
```



- Pop from stack → reverse of push
  1) Add 4 to $30 ($30 = $30 + 4)
  2) Load word 4 bytes before $30

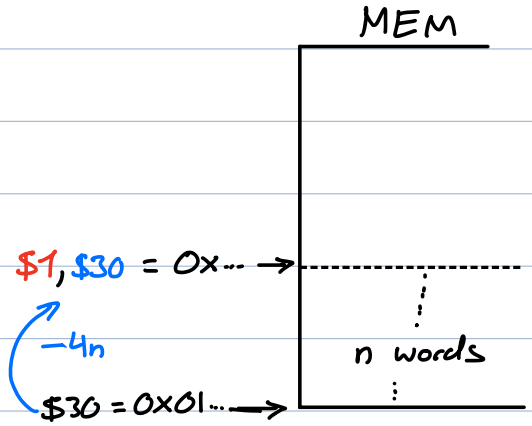
```
lis  $1
.word 4
add  $30, $30, $1
lw   $3, -4 ($30)
```



- Allocating arrays on the Stack:
  ↳ to alloc n words, decrement $30 = $30 - 4n

ie: (assuming $2 = n)

```
lis $4
.word 4
mult $2, $4
mflo $5
sub $30, $30, $5
add $1, $30, $0
```



- Store & Retrieve multiple registers from the stack

```
sw $1, -4 ($30)
sw $2, -8 ($30)
sw $4, -12 ($30)
```

} push registers you will use to the stack

```
lis $4
.word 12
sub $30, $30, $4
```

← move stack pointer

```
    ⋮

lis $4
.word 12
add $30, $30, $4
```

← update stack pointer

```
lw $1, -4 ($30)
lw $2, -8 ($30)
lw $4, -12 ($30)
```

} retrieve original values!

eg: write a procedure SumDigits that takes a positive integer in $1 & stores the result in $3.

* Ensure all registers are preserved (except $3)

↳ if $1 = 123 , $3 = 1+2+3 = 6

SumDigits :

```
; save all registers we want to use on the stack
sw   $1, -4($30)    ; push $1
sw   $4, -8($30)    ; Idea: use $4 as a temp var
sw   $10, -12($30)  ; let $10 = 10
lis  $4
.word 12
sub  $30, $30, $4   ; updates stack pointer

 ; Procedure body
lis  $10
.word 10            ; set $10 = 10
add  $3, $0, $0 ; clear $3 out, $3 = 0
loop: div  $1, $10
      mfhi $4           ; mfhi contains remainder
      add  $3, $3, $4   ; add to sum
      mflo $1           ; mflo has quotient
      bne  $1, $0, loop
```

```
; Reset all used registers to their original values
lis $4
.word 12
add $30, $30, $4    ; update stack pointer
lw $1, -4($30)      ; pop $1
lw $4, -8($30)      ; pop $4
lw $10, -12($30)    ; pop $10
jr $31              ; return!
```

Procedures

- Procedures ≈ Helper functions
  - ↳ procedure = label in front of assembly instructions
  - ↳ need to jump to this label & return some result
- Problems:
  - ↳ registers have global scope, no local scope variables
  - ↳ How will we manage caller & callee relationships?
    - ↳ How about procedures that call procedures?
      - → recursion?
  - ↳ How do we handle parameter passing & return values?
- Idea 1: use beq / bne

```
beq $0, $0, foo  ------>  foo:
                            ⋮
                          beq $0, $0, ???
```

how do we return?

- Idea 2: use jr

```
lis $3
.word foo  ; loads foo's address
jr $21
```

```
foo:
  ⋮
; How do we return?
```

Same problem as beq/bne, callee needs to know the return address!

- Idea 3: Use jalr $s
  - ↳ Sets $31 = PC & PC = $s
  - ↳ fixes return address issue!

```
lis $3
.word foo
jalr $
; How do we
; return now?
; $31's original value is lost!
```

```
foo:
  ⋮
jr $31 ; jumps to $31
```

- Solution : Use the stack & jalr
  - ↳ push $31 & all parameters onto the stack before using jalr & pop it off when call returns!
  - ↳ lets use handle caller-callee relations & recursion!

```
sw $31, -4($30)     ; Push $31 to the stack
lis $31             ; Can use $31 since it's saved
.word 4
sub $30, $30, $31
lis $3
.word foo
jalr $3
lis $31
.word 4
add $30, $30, $31
lw $31, -4($30)  ; Pop $31 from the stack
 ⋮
jr $31
```

```
foo:
 ⋮
jr $31
```

eg: Write a program that uses the SumDigits procedure to sum all digits in $1 & $2 storing the result in $3. $1 & $2 don't need to be preserved

↳ if $1 = 123 & $2 = 4567 then $3 = 28

↳ Idea: Preserve $31, call & sum the results of SumDigits on $1 & $2, restore $31

```
sw   $31, -4($30)    ; push $31
lis  $31
.word 4
sub  $30, $30, $31   ; update stack pointer

; $1 contains first input from user
lis  $3
.word SumDigits
jalr $3              ; calls SumDigits, result in $3
add  $1, $2, $0      ; get next arg for SumDigits
add  $2, $3, $0      ; Store return result from previous SumDigits

; Now that $1 has the next arg, call SumDigits again
lis  $3
.word SumDigits
jalr $3
add  $3, $2, $3      ; $2 contains result of SumDigits on initial $1
                     ; $3 contains result of SumDigits on initial $2
```

```
; $3 has our desired result! Now restore $31 & return
lis $31
.word 4
add $30, $30, $31    ; update stack pointer
lw $31, -4($30)      ; pop $31
jr $31               ; return!
... Assume SumDigits code from previous eg is here ...
```

eg: Write a program that takes a non-negative integer n in $1 & stores n! in $3. Use recursion instead of loops.

fact:

```
sw $31, -4($30)
sw $1, -8($30)
sw $11, -12($30)
lis $31
.word 12
sub $30, $30, $31

lis $11
.word 1
bne $1, $0, recur    ; If $1=0, base case where $3=1
add $3, $11, $0
beq $0, $0, clean    ; Start to unravel recursive calls!
```

```
recur :      ; calls fact with $1-1 (n-1)
 sub $1, $1, $11
 lis $31
 .word fact
 jalr $31

 add $1, $1, $11   ; restore value of n at this stack call
 mult $3, $1       ; Multiply previous answer by $1 to get
 mflo $3           ; newest factorial
```

↓ no jr, goes straight to clean!

```
clean :            ; restore current stack frame's vars
 lis $31
 .word 12
 add $30, $30, $31
 lw $31, -4($30)
 lw $1, -8($30)
 lw $11, -12($30)
 jr $31            ; ends program or returns to recur's
                   ; jalr $31
```