### Tutorial 04
- Stacks
- Procedures

#### Stacks
- $\text{SP} = \text{Stack Pointer}$
  - Initially an out of bounds address (empty)
  - Grows backwards every 4 bytes
- Pushing to the Stack
  1) Store the word 4 bytes before $\text{SP}$ ($i = -4$)
  2) set $\text{SP} = \text{SP} - 4$ (points at the top of the stack)

  **Example:** Push register $1$ onto the stack

  ```
  sw $1, -4($sp)
  ```

  - 4 bytes before $\text{SP}$

- Pop from stack \(\rightarrow\) reverse of push
  1) Add 4 to $\text{SP}$ ($\text{SP} = \text{SP} + 4$
  2) Load word 4 bytes before $\text{SP}$

  ```
  add $sp, $sp, 4
  lw $3, -4($sp)
  ```
• Allocating arrays on the Stack:

- to alloc n words, decrement $30 = $30 - 4n
- (assuming $2 = n)

```assembly
lis $4
.word 4
mult $2, $4
mul $10 $2
sub $30, $30, $5
add $1, $30, $0
```

```
MEM

\[ \text{MEM} \]

\[ \text{MEM} \]
```

• Store & Retrieve multiple registers from the stack

```assembly
sw $1, -4 (\$30)
sw $2, -8 (\$30)  \{ \text{push registers you will use to the stack} \}
sw $4, -12 (\$30)
lis $4

.word 12

sub $30, $30, $4  \{ \text{move stack pointer} \}
```

```assembly
lis $4

.word 12

add $30, $30, $4  \{ \text{update stack pointer} \}
```

```
lw $1, -4 (\$30)
lw $2, -8 (\$30)  \{ \text{retrieve original values} \}
lw $4, -12 (\$30)
```
Example: write a procedure `SumDigits` that takes a positive integer in `$1` & stores the result in `$3`.

- Ensure all registers are preserved (except `$3`)

L > if $1=123$, $3=1+2+3=6$

```assembly
SumDigits:
; Save all registers we want to use on the stack
sw $1, -4($sp)  ; push $1
sw $4, -8($sp)   ; Idea: use $4 as a temp var
sw $10, -12($sp) ; 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
       mfh $4       ; mfh contains remainder
       add $3, $2, $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, -21($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 beg/one

beg $0, $0, foo → foo:
  ...
  beg $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 beg/bre;
callee needs to know the return address!

Idea 3: Use jalr $5

```
L> Sets $31 = PC & PC = $5
L> Fixes return address issue!
lis $3
.word foo
jalr $ -> foo:
```

> How do we return now?

jr $31 ; jumps to $31

> $31’s original value is lost!
Solution: Use the stack & jalr

1. Push $31 & all parameters onto the stack before using jalr & pop it off when call returns!
2. Let's use handle caller-callee relations & recursion!

```assembly
sw $31, -4($30) ; Push $31 to the stack
lis $31        ; I can use $31 since it's saved
.word 4
sub $30, $30, $31
 lis $3
.word foo
jalr $3        ; foo:
 lis $31      ;
.word 4      ; jr $31
```

```
add $30,$30,$31
lw $31,-4($30) ; Pop $31 from the stack

jr $31
```
E.g.: 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.

L> i f $\$1 = 123$ \& $\$2 = 4567$ then $\$3 = 28$

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

sw $\$31, -4($\$30$) ; push $\$31$
li $\$3$
.word 4
sub $\$30, $\$30, $\$31$ ; update stack pointer

; $\$1$ contains first input from user
li $\$3$
.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
li $\$3$
.word SumDigits
jalr $\$3$
add $\$3, $\$2, $\$3$ ; $\$2$ contains result of SumDigits on initial $\$1$
li $\$3$ ; $\$3$ contains result of SumDigits on initial $\$2
.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)
is $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
bge $0, $0, clean  ; Start to unravel recursive calls!
recur:  i calls fact with $t-1 (n-1)

sub $t, $t, $t
lis $s1

.word fact
jair $s1

add $t, $t, $t1  ; restore value of $n at this stack call
mult $s3, $t  ; multiply previous answer by $t to get
mflo $s3  ; newest factorial

; no jr, goes straight to clean!
clean:  ; restore current stack frame's vars
lis $s1

.word 12
add $s30, $s30, $s31
lw $s31, -4($s30)
lw $s1, -8($s30)
lw $s11, -12($s30)

jr $s31  ; ends program or returns to recur's
          ;  jair $s31