

## Assembly-Level Programming

---

---

---

---

---

---

---

---

## Programming

```
for (i=0; i < 100; i++) {  
    a[i] = a[i] + 5  
}
```

```
00      AND R0, R0, #0      ; i ← 0  
01      ADD R2, R0, #-12  
02      ADD R2, R2, #-13  
03      ADD R2, R2, R2  
04      ADD R2, R2, R2      ; R2 ← -100  
05 LOOP LDR R1, R0, #30    ; R1 ← M(30+i), @a=30  
06      ADD R1, R1, #5  
07      STR R1, R0, #30    ; R1+5 → M(30+i)  
08      ADD R0, R0, #1      ; i ← i + 1  
09      ADD R3, R0, R2  
0A      BRn LOOP          ; if i < 100 goto LOOP  
0B      HALT
```

---

---

---

---

---

---

---

---

## Assembly Programming

- Assembly: an abstraction level above machine language (e.g., 1000101001100111).

```
01 ;  
02 ; Example – Multiply by 6  
03 ;  
04 .ORIG x3050  
05 LD R1, SIX  
06 LD R2, NUMBER  
07 AND R3, R3, #0 ; R3 ← 0  
08 ;  
09 ; Innermost loop  
0A ;  
0B AGAIN ADD R3, R3, R2  
0C ADD R1, R1, #-1  
0D BRp AGAIN  
0E ;  
0F HALT  
10 ;  
11 NUMBER .BLKW 1  
12 SIX .FILL x0006  
13 ;  
14 .END
```

---

---

---

---

---

---

---

---

## Assembly Syntax

Line #	Label	Opcode	Operands	Comment
--------	-------	--------	----------	---------

- Instruction syntax:
  - Line #: do not confuse with instruction address in memory
  - Labels ( $\approx$  pointers):
    - ◊ Instruction address (`BRp AGAIN`)
    - ◊ Data address (`LD R1, SIX`)
  - Opcodes: depends on ISA
  - Operands:
    - ◊ Register ( $R_i$ )
    - ◊ Address or label
- An assembly program contains
  - Instruction
  - Constant data
- Directives interpreted by assembler

---

---

---

---

---

---

---

---

---

---

## Assembly Programming - Directives

Directive	Meaning	Example
<b>.ORIG</b> <i>address</i>	Memory address of the first program instruction	<code>.ORIG x3050</code> Address of <code>LD R1, SIX</code> will be <code>0x3050</code>
<b>.BLKW</b> <i>nb</i>	The <i>nb</i> following words are not used (skipped); it is a memory allocation	<code>NUMBER .BLKW 1</code> Address <code>0x3057</code> will be used by variable <code>NUMBER</code> .
<b>.FILL</b> <i>value</i>	Store <i>value</i> at the current memory address	<code>SIX .FILL x0006</code> Data at address <code>0x3058</code> is <code>0x0006</code> .
<b>.STRINGZ</b> <i>string</i>	Store <i>character string</i> in the next words using ASCII encoding, and add 0 at the end	<code>ESSAI .STRINGZ «Hello»</code> <code>0x3059: x0048</code> <code>0x305A: x0065</code> <code>0x305B: x006C</code> <code>0x305C: x006C</code> <code>0x305D: x006F</code> <code>0x305E: x0000</code>

---

---

---

---

---

---

---

---

---

---

## Assembly

- 2 passes:
  - 1<sup>st</sup> pass: record symbol and instruction addresses
    - ◊ **Symbol table**

Symbol Name	Address
<code>AGAIN</code>	<code>3053</code>
<code>NUMBER</code>	<code>3057</code>
<code>SIX</code>	<code>3058</code>
  - 2<sup>nd</sup> pass: translate assembly instructions into machine language (binary)
    - ◊ Substitute symbols with their address, as specified in symbol table

---

---

---

---

---

---

---

---

---

---

## Assembly

- A program:
  - Is usually composed of multiple files
  - Often uses operating system routines (e.g., read/write).
  - ⇒ Link phase
- Necessary steps:
  - Merge the different code parts
  - Reference symbols from other files
  - ⇒ Their addresses are determined after link phase
  - ⇒ Requires a **.EXTERN symbol** directive
  - ⇒ The full binary code is only available after the link phase

---

---

---

---

---

---

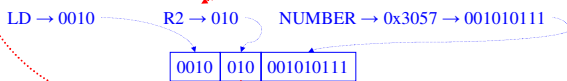
---

---

## Assembly

Address	Binary instruction	Instruction in hexa	Label	Assembly instruction
x3050	0010001001011000	x2258		LD R1, SIX
x3051	0010010001010111	x2457		LD R2, NUMBER
x3052	0101011011100000	x56E0		AND R3, R3, x0000
x3053	0001011011000010	x16C2	AGAIN	ADD R3, R3, R2
x3054	0001001001111111	x127F		ADD R1, R1, xFFFF
x3055	0000001001010011	x0253		BRP AGAIN
x3056	1111000000100101	xF025		TRAP HALT
x3057	0000000000000000	x0000	NUMBER	NOP
x3058	0000000000000110	x0006	SIX	NOP

- Multiplication program after assembly




---

---

---

---

---

---

---

---

## Example with Alpha 21264

```

.set noat
.set noreorder
.text
.arch generic
.align 4
.file 1 "test01.c"
.loc 1 2
.globl main
.ent main
main:
    lda    $sp, -400($sp)
    .frame $sp, 400, $26
    .prologue 0
    clr   $1
    mov  $sp, $2
    unop

L$2:
    ldl   $4, ($2)
    addl  $1, 1, $1
    cmplt $1, 100, $5
    lda  $2, 4($2)
    addl $4, 5, $4
    stl  $4, -4($2)
    bne  $5, L$2
    clr  $0
    lda  $sp, 400($sp)
    ret  ($26)
    
```

*Source Code*

*Assembly Code*

---

---

---

---

---

---

---

---

## Example with Alpha 21264

```
main:
0x120001100: lda    sp, -400(sp)    ; move stack pointer
0x120001104: bis    r31, r31, r1       ; r1 ← 0
0x120001108: bis    r31, sp, r2       ; r2 ← stack pointer
                                (here, beginning array a)
0x12000110c: ldq_u  r31, 0(sp)       ; NOP
0x120001110: ldl    r4, 0(r2)       ; r4 ← @(r2)
0x120001114: addl  r1, 0x1, r1    ; i ← i + 1
0x120001118: cmplt r1, 0x64, r5   ; i < 100 ?
0x12000111c: lda    r2, 4(r2)     ; r2 ← r2 + 4
0x120001120: addl  r4, 0x5, r4    ; a[i] + 5
0x120001124: stl    r4, -4(r2)    ; a[i] ← a[i] + 5
0x120001128: bne   r5, 0x120001110 ; loop if i < 100
0x12000112c: bis    r31, r31, r0   ; NOP
0x120001130: lda    sp, 400(sp)   ; restore stack pointer
0x120001134: ret    r31, (r26), 1 ; PC ← r26
```

*Machine Code*

---

---

---

---

---

---

---

---

---

---

---

---

## Data Structures

- Integer variables: 1 word
- Floating-Point variables: 2 consecutive words for LC-2
- Array:  $n$  consecutive words
- String: normally,  $n$  bytes;  $n$  words in LC-2

0x4000	x0015	var1
0x4001	x0000	var2 (least)
0x4002	x4060	var2 (most)
0x4003	x0001	var4[0]
0x4004	x0002	var4[1]
0x4005	x0003	var4[2]
0x4006	x0065	e
0x4007	x0073	s
0x4008	x0073	s
0x4009	x0061	a
0x400A	x0069	i
0x400B	x0000	\0

```
int    var1      = 21;
double var2      = 3.5;
int    var4[3]   = {1, 2, 3};
char   var5[6]   = «essai»;
```

---

---

---

---

---

---

---

---

---

---

---

---

## Passing Data Structures

- Complex data structures (e.g., arrays, strings,...)

```
MAIN
... ; address of tab in R6
LDR R0, R6, #2 ; R0←tab[2]
ADD R0, R0, #1
STR R0, R6, #2 ; R0←tab[2]
ADD R0, R6, #0 ; Compute address of tab and send to MULT

main() {
int tab[3];
...
tab[2] += 1;
MULT mult(tab);
...
}

void mult(int *tab) {
a = tab[0]
+ tab[1]
+ tab[2];
return a;
}

LDR R1, R0, #0 ; R1←tab[0]
LDR R2, R0, #1 ; R2←tab[1]
ADD R1, R2, R1
LDR R2, R0, #2 ; R1←tab[2]
ADD R1, R2, R1
...
```

---

---

---

---

---

---

---

---

---

---

---

---

## Control Structures

- Test: if (...) {...} else {...}

```

...
if (x > 5) {
    x += 1;
}
else {
    x -= 1;
}
...
END
...

LDR R0, R6, #3 ; R0=x
ADD R1, R0, #-5 ; x-5
BRnz ELSE ; x-5 ≤ 0 ?
ADD R0, R0, #1
BR END

ELSE
ADD R0, R0, #-1
...
END
...

if (x == 5) {
    x += 1;
}
else {
    x -= 1;
}
...
END
...

LDR R0, R6, #3 ; R0=x
ADD R1, R0, #-5 ; x-5
BRnp ELSE ; x-5 ≠ 0 ?
ADD R0, R0, #1
BR END

ELSE
ADD R0, R0, #-1
...
END
...

```

---

---

---

---

---

---

---

---

## Control Structures

- While loop: while (...) {...}

```

...
while (x > 5) {
    x += 1;
}
...
END
...

LOOP
LDR R0, R6, #3 ; R0=x
ADD R1, R0, #-5 ; x-5
BRnz END ; x-5 ≤ 0 ?
ADD R0, R0, #1
STR R0, R6, #3
BR LOOP

END
...

```

- For loop: for (init; test; update) {...}

```

...
for (i=0; i < 5; i++) {
    x += 1;
}
...
END
...

AND R0, R0, #0 ; i=0
STR R0, R6, #4 ; i=R0
LOOP
LDR R0, R6, #4 ; R0=i
ADD R1, R0, #-5 ; i-5
BRzp END ; i-5 ≥ 0 ?
LDR R0, R6, #3 ; R0=x
ADD R0, R0, #1
STR R0, R6, #3 ; R0=x
LDR R0, R6, #4 ; R0=i
ADD R0, R0, #1
STR R0, R6, #4 ; R0=i
BR LOOP

END
...

```

---

---

---

---

---

---

---

---

## Function Calls

```

.ORIG x3000
LD R0, A
LD R1, B
JSR MULT
ADD R5, R2, #0
LD R0, C
LD R1, D
JSR MULT
ADD R5, R5, R2
HALT
; --- Data
A .FILL x0002
B .FILL x0002
C .FILL x0002
D .FILL x0003
; --- Multiplication subroutine
MULT AND R2, R2, #0
LOOP ADD R2, R2, R1
ADD R0, R0, #-1
BRp LOOP
RET
.END

```

- Use registers to pass parameters:
  - Fast
  - Limited number of parameters
- Return values:
  - One word (return)
  - Convention: for instance, use register R2
- Don't forget:
  - Return address in R7

---

---

---

---

---

---

---

---

## Function Calls

```

.ORIG x3000
LD R0, A
LD R1, B
JSR MULT
ADD R5, R2, #0
LD R0, C
LD R1, D
ST R5, BAK_R5
JSR MULT
LD R5, BAK_R5
ADD R5, R5, R2
HALT
; --- Sauvegarde
BAK_R5 .BLKW 1
; --- Donnees
A .FILL x0002
B .FILL x0002
C .FILL x0002
D .FILL x0003
; --- Procedure de multiplication
MULT AND R2, R2, #0
LOOP ADD R2, R2, R1
ADD R0, R0, #-1
BRp LOOP
RET

```

- Registers used in subroutine often **unknown**:
  - Store registers before function call
- Or
  - Store modified register in called function
- Alternative: use memory to pass parameters

---

---

---

---

---

---

---

---

## Function Calls Stack

- For a function call, one must:
  - Save registers modified by the caller
  - Pass parameters through memory
  - Return result
  - Have memory locations for local variables
 ⇒ function **context**
- Dynamically allocated memory area: the **stack**.
- Note: For just a few values, it is still faster to use registers rather than the stack to pass values

---

---

---

---

---

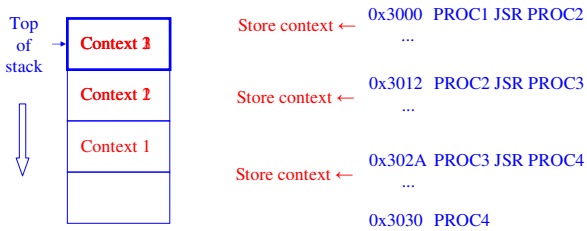
---

---

---

## Stack - Principle

- LIFO (*Last In First Out*)
- Only latest context visible and accessible



- In reality:
  - The stack can correspond to a large memory chunk
  - Contexts are naturally not moved
  - There is a stack pointer to indicate the top of the stack or the first available memory location (usually a register)
  - Sometimes (as in LC-2) a pointer to indicate beginning of a context
- LC-2 designers selected stack pointer: R6, and context pointer: R5

---

---

---

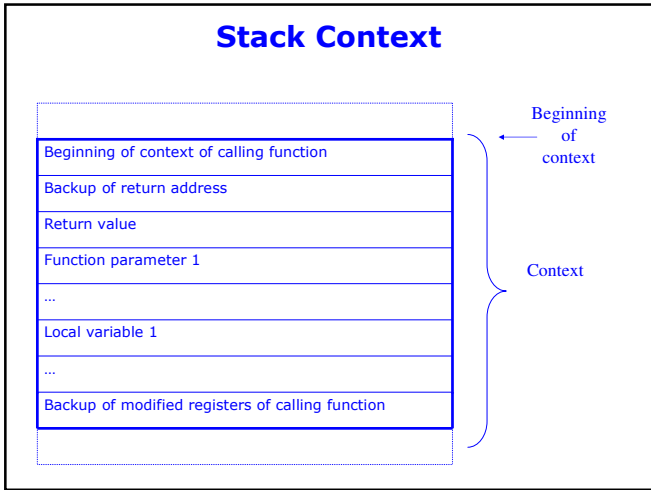
---

---

---

---

---




---

---

---

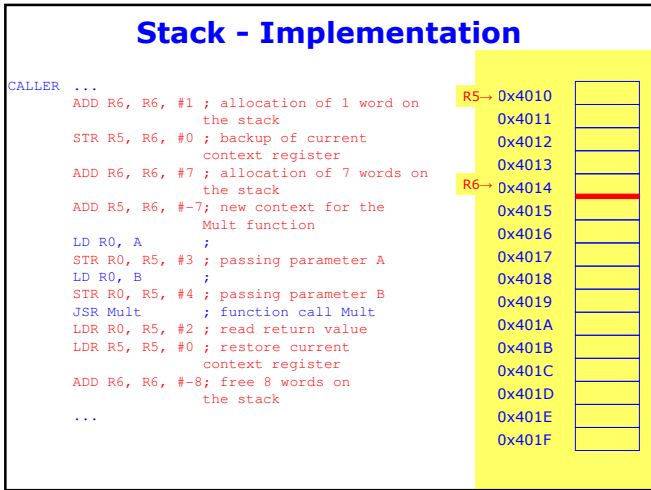
---

---

---

---

---




---

---

---

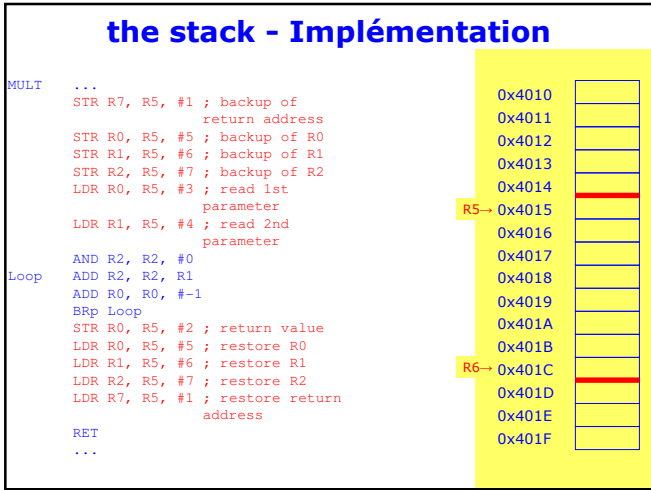
---

---

---

---

---




---

---

---

---

---

---

---

---

## Recursive Function Calls

```

Sum (int n) {
    if (n > 1) return (n + Somme(n-1));
    else return 1;
}
SUM  STR R7, R5, #1 ; backup of return address
      STR R0, R5, #4 ; backup of R0
      STR R1, R5, #5 ; backup of R1
      LDR R0, R5, #3 ; read n on the stack
      ADD R1, R0, #-1 ; n > 1 ?
      BRnz END
      ADD R6, R6, #1 ; allocation of 1 word
      STR R5, R6, #0 ; backup current
                        context register
      ADD R6, R6, #5 ; allocation of 6 words
      ADD R5, R6, #-5 ; new context
      STR R1, R5, #3 ; pass n-1 as a parameter
      JSR SUM
      LDR R1, R5, #2 ; read return value
      ADD R0, R1, R0 ; Sum(n-1) + n
      LDR R5, R5, #0 ; restore current
                        context register
      ADD R6, R6, #-6 ; free 6 words on the stack
END   STR R0, R5, #2 ; store return value
      LDR R0, R5, #4 ; restore R0
      LDR R1, R5, #5 ; restore R1
      LDR R7, R5, #1 ; restore return address
      RET

```

R5→	0x4010	
	0x4011	
	0x4012	
	0x4013	
	0x4014	
R6→	0x4015	
	0x4016	
	0x4017	
	0x4018	
	0x4019	
	0x401A	
	0x401B	
	0x401C	
	0x401D	
	0x401E	
	0x401F	

