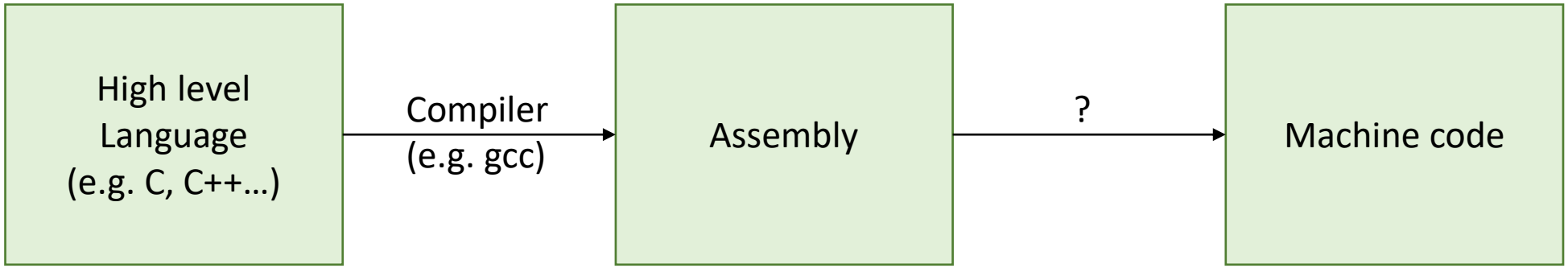
Computation structures
Tutorial: β-assembly (part 1)

Programming a computer

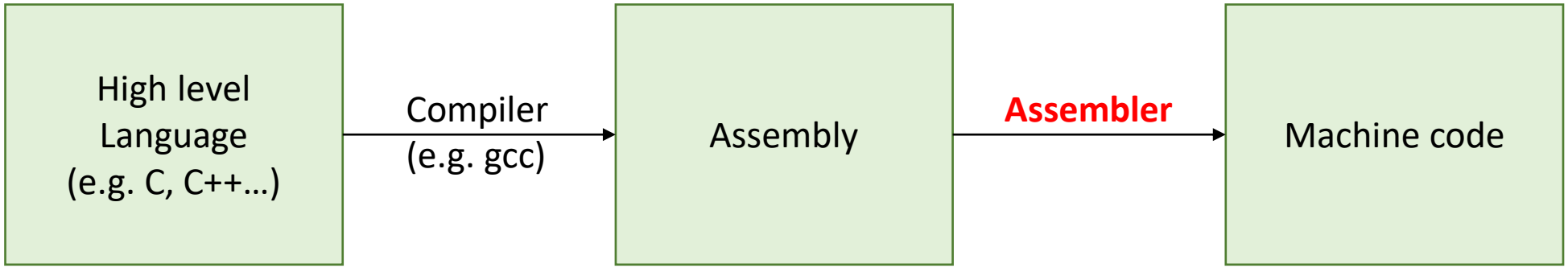



```
int tmp = *a;
*a = *b;
*b = *a;
```

```
LD(R1, 0, R3)
LD(R2, 0, R4)
ST(R4, 0, R1)
ST(R3, 0, R2)
```

```
0x00 0x00 0x61 0x60
0x00 0x00 0x82 0x60
0x00 0x00 0x81 0x64
0x00 0x00 0x62 0x64
```

Programming a computer



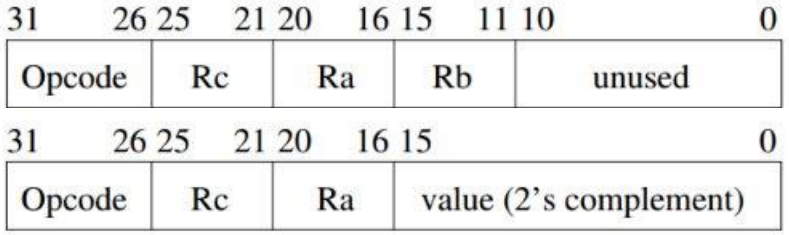

```
int tmp = *a;
*a = *b;
*b = *a;
```

```
LD(R1, 0, R3)
LD(R2, 0, R4)
ST(R4, 0, R1)
ST(R3, 0, R2)
```

```
0x00 0x00 0x61 0x60
0x00 0x00 0x82 0x60
0x00 0x00 0x81 0x64
0x00 0x00 0x62 0x64
```

How to program the β-machine ?

- Instruction of the β-machine:




```
e.g.: ADD(R1, R2, R3) -> 0x80611000
      LD(R1, 16, R2)  -> 0x60410010
```

- **Programming** the machine = **writing** a sequence of such **instructions**
- How to avoid writing instructions directly ?
- We need an **assembler** to convert symbolic notations into instructions !

Let us define the **β-assembler**

A first program in β-assembly

- The **input** of the assembler is a **sequence of constant expressions**
- The **output** is a **sequence of bytes**

```
Input : 0x25 0x35 0x16+0x2
        3>>2 3+1 35>>2
        0x2500

Output: 0x25 0x35 0x18
        0x01 0x04 0x12
        0x00
```

- **Still too low level** so need higher level mechanisms

Identifiers

- An **identifier** can be seen as a variable
- It can be assigned a value or used in expressions
- **Special identifier « . »** is the **position of the next byte** to be added to the sequence

```
Input : r = 3
        r r+1 r*2 r-(r/2)
        . = . + 4   | 4 bytes are skipped
        0x25 r<<2

Output: 0x03 0x04 0x06 0x02
        0x00 0x00 0x00 0x00
        0x25 0x0C
```

Labels

- A position can be assigned an identifier

```
Input: r = .
       0x25 0x33 r
```

- Or equivalently as a **label**:

```
Input: r: 0x25 0x33 r
```

```
Output: 0x25 0x33 0x00
```

Macros

- A macro is a **parametrized program fragment**
- Macro **definition**:

```
.macro macro_name(p1, …, pn) body
.macro macro_name(p1, …, pn) {
    body
}
```

- Macro **invocation**:

```
macro_name(v1, …, vn)
```

Example:

```
Input : .macro F(x) x+1 x+2 x*4
        .macro G(x) { F(x) F(x+2) }
        G(1) F(0)

Output: 0x02 0x03 0x04 0x04
        0x05 0x0C 0x01 0x02
        0x00
```

That’s all folks !

- That is all the language constructs we need for writing programs
- However, a question remains:

How to generate machine code for the β-machine using such constructs ?

That’s all folks !

- That is all the language constructs we need for writing programs
- However, a question remains:

How to generate machine code for the β-machine using such constructs ?

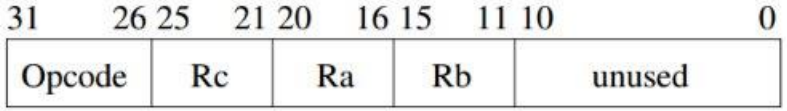
We have to define the micro-assembly !

β-assembly, micro-assembly for the β-machine

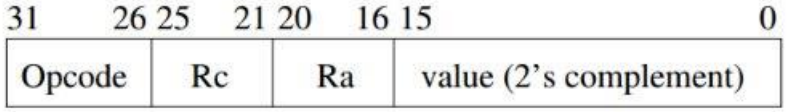
- Let us define the **β-assembly** (see `beta.uasm`)
- **That is:** a set of macro making easy writing instructions for the machine
- **Note**: always start to write least significant byte

```
.macro WORD(x) x%0x100 (x>>8)%0x100
.macro LONG(x) WORD(x) WORD(x >> 16)
```

ENC_NOLIT:



ENC_LIT:




```
.macro ENC_NOLIT(OP,RA,RB,RC) LONG((OP<<26)+((RC%0x20)<<21)+((RA%0x20)<<16)+((RB%0x20)<<11))
.macro ENC_LIT(OP,RA,CC,RC) LONG((OP<<26)+((RC%0x20)<<21)+((RA%0x20)<<16)+(CC%0x10000))
.macro ENC_ADRLIT(OP,RA,RC,label) ENC_LIT(OP, RA, RC, (label - (. + 4)) >> 2 )
```

Defining addresses of registers as identifiers

```
| Registers
r0 = 0b0
r1 = 0b1
| …
r31 = 0b11111
bp = 27                 | frame pointer (points to base of frame)
lp = 28                 | linkage register (holds return adr)
sp = 29                 | stack pointer (points to 1st free locn)
xp = 30                 | interrupt return pointer (lp for interrupts)

R0 = r0
R1 = r1
| …
R31 = r31
BP = bp
lp = LP
```

β-assembly – actual instructions

Without a literal		With a literal	
Opcode	name	Opcode	name
$0x20$	ADD	$0x30$	ADDC
$0x21$	SUB	$0x31$	SUBC
$0x22$	MUL	$0x32$	MULC
$0x23$	DIV	$0x33$	DIVC

Opcode	name
$0x1D$	BEQ/BF
$0x1E$	BNE/BT

```
|; ADD(Ra, Rb, Rc)      Reg[Rc] <- Reg[Ra] + Reg[Rb]

.macro ADD(Ra,Rb,Rc)     ENC_NOLIT(0x20,Ra,Rb,Rc)

|; BEQ(Ra, label, Rc)  Reg[Rc] <- PC; if Reg[Ra] == 0 then PC <- Mem[PC + CC]

.macro BEQ(Ra,label,Rc) ENC_ADRLIT(0x1D,Ra,Rc,label)

|; MOVE(Ra, Rb)         Reg[Rb] <- Reg[Ra]

.macro MOVE(Ra,Rb)       ADD(Ra,R31,Rb)
```

β-assembly – Writing a program

As simple as defining identifiers and invoking macros:

```
   .include beta.uasm  |; Include the definition of beta-assembly

   main:
0       CMOVE(0x25, R1)   |; Reg[R1] <- 0x25
4       CMOVE(0x876, R2)  |; Reg[R2] <- 0x876
8       ADD(R1, R2, R3)   |; Reg[R3] <- Reg[R1] + Reg[R2]
12      MULC(R3, 4, R3)   |; Reg[R3] <- Reg[R3] * 4
16      LD(R3, 0, R4)     |; Reg[R4] <- Mem[Reg[R3] + 0]
```

β-assembly – Functions

For supporting **function/procedure** calls we need:

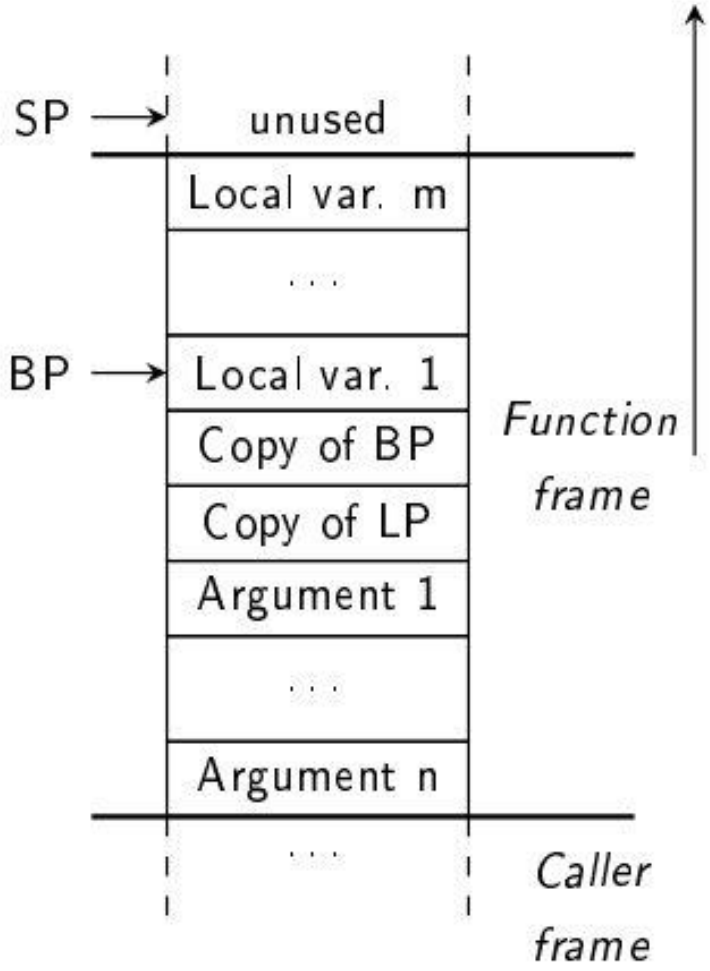
- Allowing procedure to **return a value**

} Use **R0 to store returned value**

- **Passing arguments** to the procedures
- Allowing a **procedure to call other procedures**, including itself
- Making it possible for a procedure to use **local variables**

} Use a **stack** !

β-assembly – Stack



Special registers:

- **BP (r27):** base of frame pointer
- **SP (r28):** stack pointer
- **LP (r29):** linkage pointer

The **frame** is the stack area used by a procedure.

LP contains the address of the instruction to branch to when the execution of a procedure is over.

SP contains the address of the first free element on top of the stack

β-assembly – How to call a procedure

In the calling procedure:

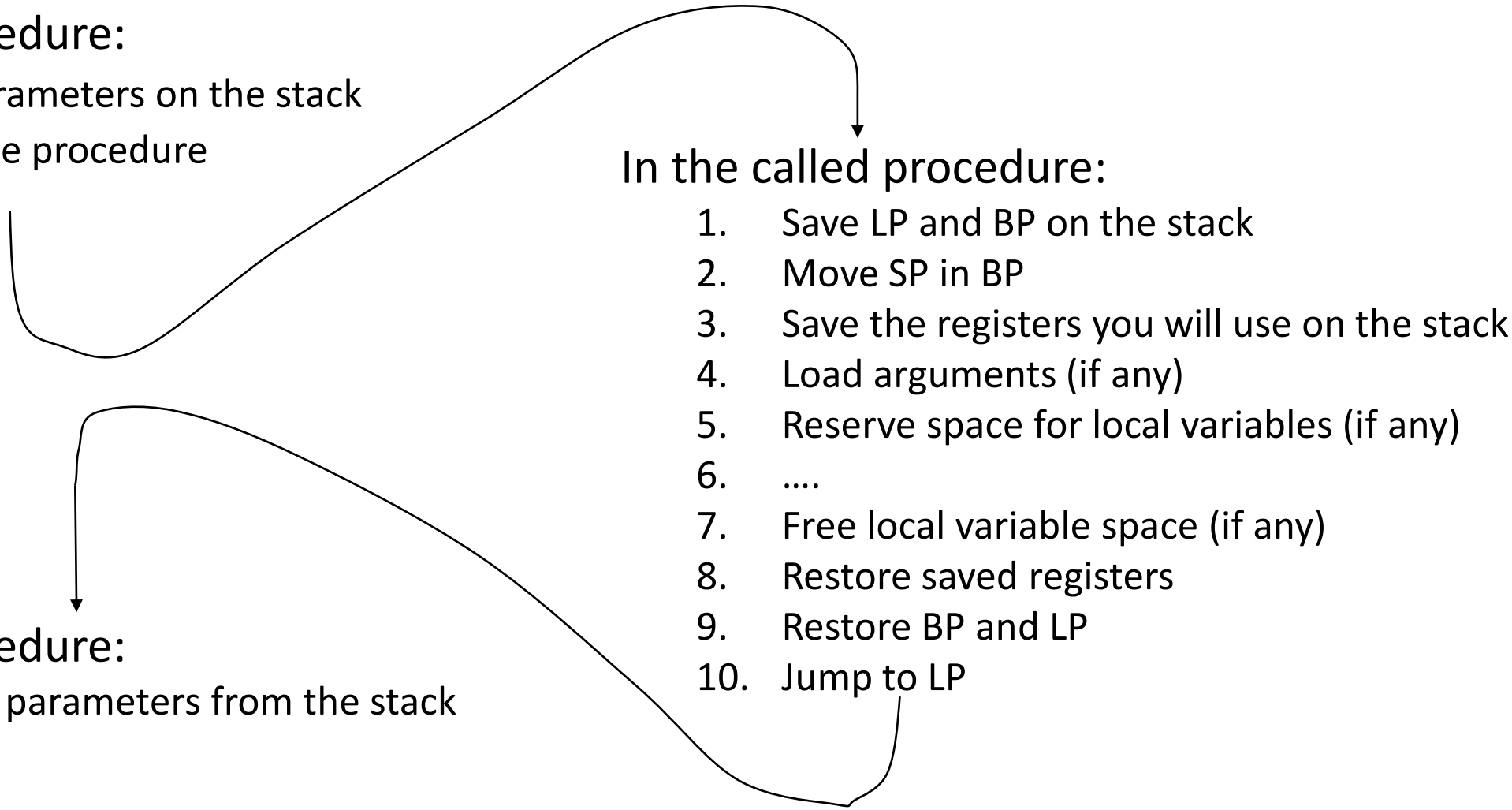
1. Push the parameters on the stack
2. Branch to the procedure

In the called procedure:

1. Save LP and BP on the stack
2. Move SP in BP
3. Save the registers you will use on the stack
4. Load arguments (if any)
5. Reserve space for local variables (if any)
6. ….
7. Free local variable space (if any)
8. Restore saved registers
9. Restore BP and LP
10. Jump to LP

In the calling procedure:

1. Remove the parameters from the stack



Useful macros

Branching

```
BR(label, Rc)   -> BEQ(R31, label, Rc)
BR(label)       -> BEQ(R31, label, R31)
BT(Ra, label)   -> BNE(Ra, label, R31)
BF(Ra, label)   -> BEQ(Ra, label, R31)
```

Procedure

```
RTN()           -> JMP(LP, R31)
CALL(label)     -> BEQ(R31, label, LP)
CALL(label, n)  -> BEQ(R31, label, LP)
                   DEALLOCATE(n)
```

Stack

```
PUSH(Ra)        -> ADDC(SP,4,SP) ST(Ra,-4,SP)
POP(Ra)         -> LD(SP,-4,Ra) SUBC(SP, 4, SP)
ALLOCATE(N)     -> ADDC(SP, 4*N, SP)
DEALLOCATE(N)   -> SUBC(SP, 4*N, SP)
```

Assignment

```
MOVE(Ra, Rb)    -> ADD(R31, Ra, Rb)
CMOVE(N, Ra)    -> ADCC(R31, N, Ra)
```

Comparison

CMP{LT|EQ|LE}[C] -> only {<, ==, <=} with or without constant !!

/!\ False friends !

LD(Ra, Rb, Rc) does not exist (if you want such operation you should define a new macro yourself) !

CMPGE/CMPGEC do not exist neither !