How can the machine ULg01 be programmed?

Summary

1. How can writing "bits" be avoided with the help of an assembly language.
2. How can high level languages be implemented on the β architecture?
3. What needs to be added to the machine ULg01 for it to able to run an operating system?

A program for the β architecture

```
10000000101000110010000000000000
01100100101100000000000000100100
```

- Impossible to read; impossible to write (for humans).
- We need something else:

```
ADD(r3,r4,r5)
ST(r5,0x24,r16)
```

- For this to be possible, we need a program that can translate from a symbolic notation to a sequence of bits: an assembler.

A Universal Assembly Language

- Rather than writing an assembler specific to the β architecture, we are going to define a few general mechanisms that make it possible to handle a wide variety of assembly languages.
- What the assembler produces is a sequence of bytes, for example

  ```
  0x80   0x64   0x28   0x00
  ```

- The input language for the assembler is a sequence of constant expressions,

  ```
  128 80+20 2*0x14 0
  ```

  A constant prefixed with `0x` is interpreted an hexadecimal value; prefixed with `0b` it is interpreted as a binary value; and, without a prefix as a decimal value.
- We will now introduce a few mechanisms that will facilitate writing sequences of expressions.

Assembly language mechanisms

- Definitions of the following form are allowed

  identifier = *expression*

  where an *identifier* is a string of letters and digits of arbitrary length starting with a letter. Once defined an identifier may appear in an expression. One can then write

```
X = 128
X X-28 2*0x14 X-X
```

- A special *identifier* "." is used to represent the position of the next byte to be added to the sequence being generated. Initially, "." has value 0 ; The value of "." can be modified, for example (read column by column)

```
X = 128 | .  = .+ 4
X X-28  | 2*0x14 X-X
```

  leaves a 4 byte space following the two first generated bytes.

- A position can be assigned to an identifier by writing

  identifier :

  This is equivalent to writing

  identifier = .

  Example :

```
X: .  = .+4
0xA6
```

  reserves (for example for a variable) a 4 byte space at the position represented by the identifier `X`.

A powerful mechanism: macros

- Macros make it possible to define a parameterized program fragment that can be reused at will. The definition of a macro has the form

  `.macro` *name(p1,. . . ,pn) body*

  where *p1,. . . ,pn* are the (formal) parameters of the macro. There can be any number of them. The *body* of the macro is a program (a sequence of expressions) in which the formal parameters may appear.

- A call to a macro is written

  name(v1,. . . ,vn)

  and is replaced by the body of the macro in which the formal parameters are replaced by the values of the expressions *v1,. . . ,vn*. Parameters are evaluated once, when the macro is called.

- A macro generating four consecutive values can be defined by

```
.macro consec(n) n n+1 n+2 n+3
```

  and thus `consec(6)` is assembled into 6 7 8 9.

Macros defining the β assembler

First some general useful macros.

```
.macro WORD(x)  (x)%256   (x)/256        | little endian
.macro LONG(x)  WORD(x) WORD((x) >> 16)
```

Instruction coding macros

```
.macro ENC_NOLIT(OP, Ra, Rb, Rc) LONG((OP<<26)+((Rc%32)<<21)
                                      +((Ra%32)<<16)+((Rb%32)<<11))
.macro ENC_LIT(OP, Ra, Rc, Lit)  LONG((OP<<26)+((Rc%32)<<21)
                                      +((Ra%32)<<16)+(Lit % 0x10000))
.macro ENC_ADRLIT(OP, Ra, Rc, Label)  ENC_LIT(OP, Ra, Rc, (label-(.+4))>>2)
```

Assigning identifiers to registers

```
r0 = 0x0
...
r31 =  0x11111
```

Logical and Arithmetical instructions

```
.macro ADD(Ra,Rb,Rc)           ENC_NOLIT(0b100000,Ra,Rb,Rc)
.macro ADDC(Ra,Lit,Rc)         ENC_LIT(0b110000,Ra,Rc,Lit)
.macro SUB(Ra,Rb,Rc)           ENC_NOLIT(0b100001,Ra,Rb,Rc)
.macro SUBC(Ra,Lit,Rc)         ENC_LIT(0b110001,Ra,Rc,Lit)
.macro DIV(Ra,Rb,Rc)           ENC_NOLIT(0b100011,Ra,Rb,Rc)
.macro DIVC(Ra,Lit,Rc)         ENC_LIT(0b110011,Ra,Rc,Lit)
.macro MUL(Ra,Rb,Rc)           ENC_NOLIT(0b100010,Ra,Rb,Rc)
.macro MULC(Ra,Lit,Rc)         ENC_LIT(0b110010,Ra,Rc,Lit)


.macro CMPEQ(Ra,Rb,Rc)         ENC_NOLIT(0b100100,Ra,Rb,Rc)
.macro CMPEQC(Ra,Lit,Rc)       ENC_LIT(0b110100,Ra,Rc,Lit)
.macro CMPLE(Ra,Rb,Rc)         ENC_NOLIT(0b100110,Ra,Rb,Rc)
.macro CMPLEC(Ra,Lit,Rc)       ENC_LIT(0b110110,Ra,Rc,Lit)
.macro CMPLT(Ra,Rb,Rc)         ENC_NOLIT(0b100101,Ra,Rb,Rc)
.macro CMPLTC(Ra,Lit,Rc)       ENC_LIT(0b110101,Ra,Rc,Lit)
```

```
.macro AND(Ra,Rb,Rc)          ENC_NOLIT(0b101000,Ra,Rb,Rc)
.macro ANDC(Ra,Lit,Rc)        ENC_LIT(0b111000,Ra,Rc,Lit)
.macro OR(Ra,Rb,Rc)           ENC_NOLIT(0b101001,Ra,Rb,Rc)
.macro ORC(Ra,Lit,Rc)         ENC_LIT(0b111001,Ra,Rc,Lit)
.macro XOR(Ra,Rb,Rc)          ENC_NOLIT(0b101010,Ra,Rb,Rc)
.macro XORC(Ra,Lit,Rc)        ENC_LIT(0b111010,Ra,Rc,Lit)

.macro SHR(Ra,Rb,Rc)          ENC_NOLIT(0b101101,Ra,Rb,Rc)
.macro SHRC(Ra,Lit,Rc)        ENC_LIT(0b111101,Ra,Rc,Lit)
.macro SHL(Ra,Rb,Rc)          ENC_NOLIT(0b101100,Ra,Rb,Rc)
.macro SHLC(Ra,Lit,Rc)        ENC_LIT(0b111100,Ra,Rc,Lit)
.macro SRA(Ra,Rb,Rc)          ENC_NOLIT(0b101110,Ra,Rb,Rc)
.macro SRAC(Ra,Lit,Rc)        ENC_LIT(0b111110,Ra,Rc,Lit)
```

Memory access instructions

```
.macro LD(Ra,Lit,Rc)           ENC_LIT(0b011000,Ra,Rc,Lit)
.macro LDR(label,Rc)           ENC_ADRLIT(0b011111,0,Rc,label)
.macro ST(Rc,Lit,Ra)           ENC_LIT(0b011001,Ra,Rc,Lit)
```

Branch instructions

```
.macro BNE(Ra,label,Rc)        ENC_ADRLIT(0b011110,Ra,Rc,label)
.macro BT(Ra,label,Rc)         BNE(Ra,label,Rc)
.macro BEQ(Ra,label,Rc)        ENC_ADRLIT(0b011101,Ra,Rc,label)
.macro BF(Ra,label,Rc)         BEQ(Ra,label,Rc)
.macro JMP(Ra,Rc)              ENC_LIT(0b011011,Ra,Rc,0)
```

Defining new instructions with macros

Examples

```
.macro MOVE(Ra,Rc)      ADD(Ra,r31,Rc)    | R[Rc] <- R[Ra]
.macro CMOVE(C,Rc)      ADDC(r31,C,Rc)    | R[Rc] <- C

.macro NOP()            ADDC(r31,r31,r31) | do nothing

.macro BR(label)        BEQ(r31,label,r31)
```

A program written in β assembler

```
.include macros              | Loading the macros
       LDR(input,r0)         | data in r0
       BR(bitrev)
input: LONG(0x12345)

bitrev:
       CMOVE(32,r2)         | 32 bits to process
       CMOVE(0,r1)          | result register
loop:  ANDC(r0,1,r3)        | first bit to r3
       SHLC(r1,1,r1)        | transfer it
       OR(r3,r1,r1)         | to r1
       SHRC(r0,1,r0)        | prepare following bit
       SUBC(r2,1,r2)        | decrement number of bits to process
       BNE(r2,loop)         | loop if not done
```

The β architecture and high-level languages

We will see how the concepts used in high-level languages, in particular, procedure calls, can be implemented in the β architecture.

Let us look at an example.

```
int fact(int n)
{
    if (n>0)
        return n*fact(n-1);
    else
        return 1;
}

fact(4);
```

To handle procedures solutions to the following problems are needed:

1. passing the value of the arguments to the procedure,
2. making it possible for the procedure to use local variables,
3. allowing the procedure to return a value,
4. allowing the procedure to call other procedures, including itself (recursion).

Saving the return address

- To save the return address, it is natural to use a register. Register r28 (Linkage Pointer, LP) will be reserved for this purpose.
- A procedure can thus be called with the instruction `BR(label,LP)`.
- And returning from a procedure is done with `JMP(LP,r31)`.
- But, how can one handle arguments and recursion?

Managing a stack in the β architecture

- A stack is a very useful concept for handling procedures as well as other programming problems.
- A stack is an area of memory in which only the most recently added element can be accessed directly.
- We will implement a stack as a sequence of memory words and we will use a dedicated register (`SP = r29`) as Stack Pointer.

- Adding or removing an element of the stack can thus be done with the following instructions.

```
.macro PUSH(Ra)         ADDC(SP,4,SP) ST(Ra,-4,SP)
.macro POP(Ra)          LD(SP,-4,Ra)  ADDC(SP,-4,SP)
```

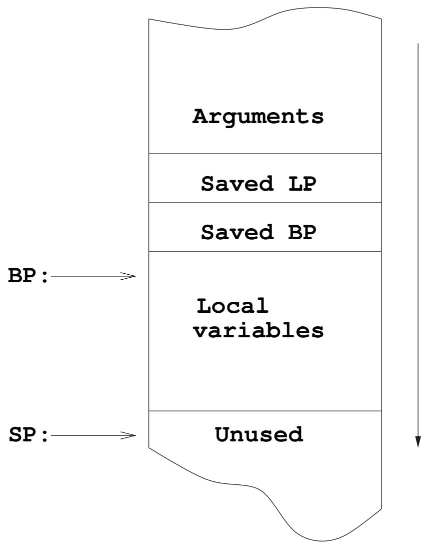
- Reserving space on the stack is done as follows.

```
.macro ALLOCATE(k)      ADDC(SP,4*k,SP)
.macro DEALLOCATE(k)    SUBC(SP,4*k,SP)
```

Handling procedures using a stack

- Arguments are placed on the stack before calling the procedure.
- The procedure uses the stack area beyond the arguments for its own needs.
- The stack area used by a procedure is called its *frame*. It is useful to have a pointer that lets us know where this frame starts (Base of frame Pointer, `BP = r27`).
- To allow recursion, LP and BP are saved on the stack at each procedure call.

General stack organisation



Implementing procedure calls

In the calling procedure

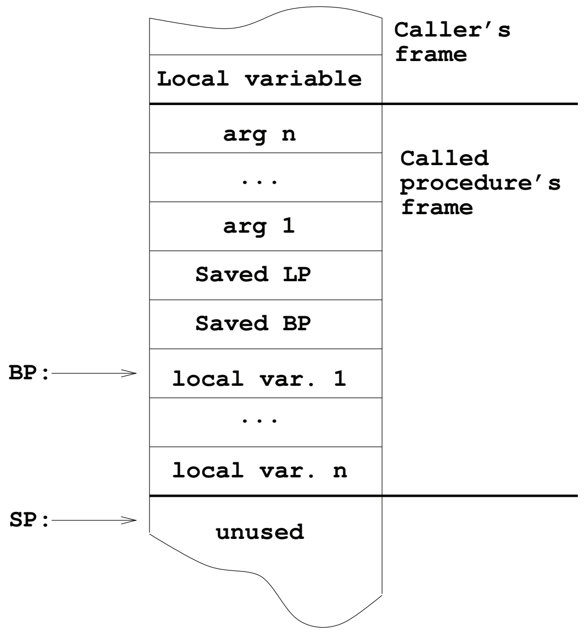
```
    PUSH(argn)     | arguments are placed in reverse
    ...            | order
    PUSH(arg1)
    BR(f,LP)
    DEALLOCATE(n)
```

At the beginning of the called procedure

```
f:  PUSH(LP)
    PUSH(BP)
    MOVE(SP,BP)
    ALLOCATE(space) | allocate local space
    (save other registers if needed)
```

at the end of the called procedure

```
    (restore saved registers)
    MOVE(val,r0)  | the returned value is placed in r0
    MOVE(BP,SP)   | restore the caller's SP
    POP(BP)       | restore the caller's BP
    POP(LP)       | restore the return address
    JMP(LP,R31)   | return
```



Accessing local variables

To access the i-th local variable, one uses

```
LD(BP,(i-1)*4,rx)

ST(rx,(i-1)*4,BP)
```

To access the j-th argument, one uses

```
LD(BP,-4*(j+2),rx)

ST(rx,-4*(j+2),BP)
```

An implementation of the factorial function

```
fact: PUSH(LP)          | save the return address
      PUSH(BP)          | save the previous frame
      MOVE(SP,BP)       | initialise the current frame
      PUSH(r1)          | r1 will be used, save it
      LD(BP,-12,r1)     | load the argument n in r1
      BNE(r1,big)       | compare n to 0
      ADDC(r31,1,r0)    | n=0, return 1
      BR(rtn)           | go to return sequence

big:  SUBC(r1,1,r1)     | compute n-1 in r1
      PUSH(r1)          | place argument on stack
      BR(fact,LP)       | recursive callm
      DEALLOCATE(1)     | free area used for arguments
      LD(BP,-12,r1)     | load n in r1
      MUL(r1,r0,r0)     | n*fact(n-1) in r0
```

```
rtn:  POP(r1)           | restore r1
      MOVE(BP,SP)       | restore  SP
      POP(BP)           | restore  BP
      POP(LP)           | restore the return address
      JMP(LP,r31)       | return
```

Non local variables

- The technique used so far does not handle non local variables.
- For this, one uses “static links”.
- A static link points from the current stack frame to the stack frame of the procedure in which the current procedure was (lexically) defined.
- One then accesses a non local variable by going up through static links the required number of times and then accessing a local variable.