
The **DLX** Instruction Set Architecture

Maurizio Palesi

Summary

- Introduction
- Registers
 - GPR
 - FPR
 - Miscellaneous
- Data format
- Addressing
- Instruction types
 - I-type
 - R-type
 - J-type
- Examples

DLX Architecture Overview

- Pronounced *delux*
- (AMD 29K, DECstation 3100, HP 850, IBM 801, Intel i860, MIPS M/120A, MIPS M/1000, Motorola 88K, RISC I, SGI 4D/60, SPARCstation-1, Sun-4/110, Sun-4/260)/13 = 560 = DLX
- Simple Load/Store architecture
- Functions that are used less often are considered less critical in terms of performances
 - Not implemented directly in DLX

DLX Architecture Overview

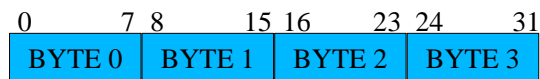
- Three architectural concepts:
 - Simplicity of load/store IS
 - Importance of pipelining capability
 - Easily decoded IS
- Stuff
 - 32 GPRs & 32 spFPRs (shared with 16 dpFPRs)
 - Miscellaneous registers
 - ✓ interrupt handling
 - ✓ floating-point exceptions
 - Word length is 32 bits
 - Memory byte addressable, Big Endian, 32-bit addr

Registers

- The DLX ISA contains 32 (R0-R31) 32-bit general-purpose registers
- Register R1-R31 are true GP registers (R0 hardwired to 0)
- R0 always contains a 0 value & cannot be modified
 - `ADDI r1,r0,imm ; r1=r0+imm`
- R31 is used for remembering the return address for JAL & JALR instructions

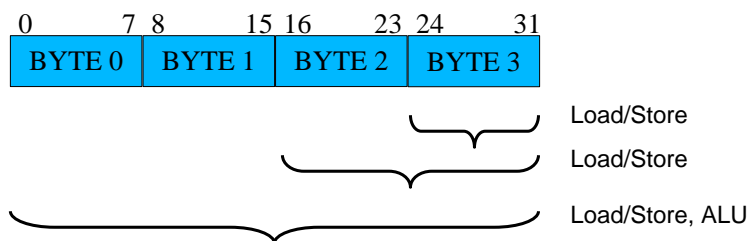
Registers

- Register bits are numbered 0-31, from back to front (0 is MSB, 31 is LSB).
- Byte ordering is done in a similar manner



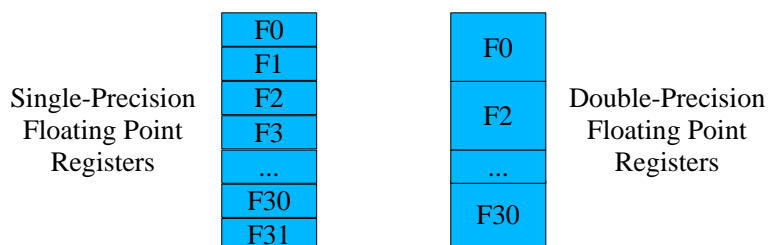
- A register may be loaded with
 - A byte (8-bit)
 - An halfword (16-bit)
 - A fullword (32-bit)

Registers



Floating-Point Registers

- 32 32-bit single-precision registers (F0, F1, ..., F31)
- Shared with 16 64-bit double-precision registers (F0, F2, ..., F30)
- The smallest addressable unit in FPR is 32 bits



Miscellaneous Registers

- There are 3 miscellaneous registers
 - **PC**, Program Counter, contains the address of the instruction currently being retrieved from memory for execution (32 bit)
 - **IAR**, Interrupt Address Register, maintains the 32-bit return address of the interrupted program when a *TRAP* instruction is encountered (32 bit)
 - **FPSR**, Floating-Point Status Register, provide for conditional branching based on the result of FP operations (1 bit)

Data Format

- Byte ordering adheres to the Big Endian ordering
 - The most significant byte is always in the lowest byte address in a word or halfword

`mem[0] ← 0xAABBCCDD`

Big Endian	byte address	Little Endian
DD	3	AA
CC	2	BB
BB	1	CC
AA	0	DD

Addressing

- Memory is byte addressable
 - Strict address alignment is enforced
- Halfword memory accesses are restricted to even memory address
 - `address = address & 0xffffffe`
- Word memory accesses are restricted to memory addresses divisible by 4
 - `address = address & 0xffffffc`

Instruction Classes

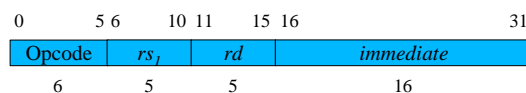
- The instructions that were chosen to be part of DLX are those that were determined to resemble the MFU (and therefore performance-critical) primitives in program
- **92** instructions in **6** classes
 - Load & store instructions
 - Move instructions
 - Arithmetic and logical instructions
 - Floating-point instructions
 - Jump & branch instructions
 - Special instructions

Instruction Types

- All DLX instructions are 32 bits and must be aligned in memory on a word boundary
- 3 instruction formats
 - **I-type** (Immediate): manipulate data provided by a 16-bit field
 - **R-type** (Register): manipulate data from one or two registers
 - **J-type** (Jump): provide for the execution of jumps that do not use a register operand to specify the branch target address

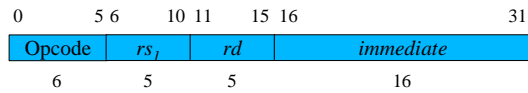
I-type Instructions (1 of 3)

- Load/Store (u/s byte, u/s halfword, word)
- All immediate ALU operations
- All conditional branch instructions
- JR, JALR



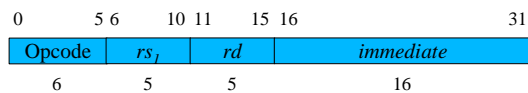
- **Opcode**: DLX instruction is being executed
- **rs1**: source for ALU, base address for Load/Store, register to test for conditional branches, target for JR & JALR

I-type Instructions (2 of 3)



- ***rd***: destination for Load and ALU operations, source for Store.
 - Unused for conditional branches and JR and JALR
- ***immediate***: offset used to compute the address for loads and stores, operand for ALU operations, sign-ext offset added to PC to compute the branch target address for a conditional branch.
 - Unused for JR and JALR

I-type Instructions (3 of 3)



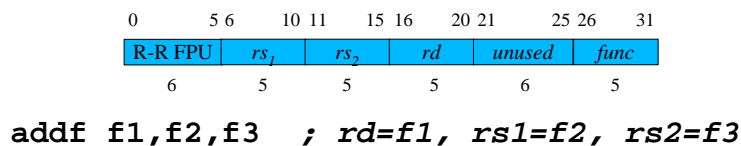
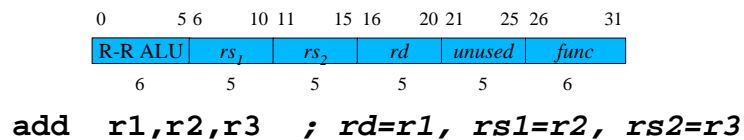
```

addi r1,r2,5      ; r1=r2+sigext(5)
                  ; rd=r1, rs1=r2, imm=0000000000000101
addi r1,r2,-5     ; r1=r2+sigext(-5)
                  ; rd=r1, rs1=r2, imm=1111111111111011
jr   r1           ; rs1=r1
jalr r1          ; rs1=r1

lw  r3, 6(r2)    ; r3=Mem[sigext(6)+r2]
                  ; rd=r3, rs1=r2, imm=6
sw  -7(r4),r3    ; Mem[sigext(-7)+r4]=r3
                  ; rd=r3, rs1=r4, imm=-7
beqz r1,target   ; if (r1==0) PC=PC+sigext(target)
                  ; rs1=r1, imm=target
jr   r1         ; PC=r1
                  ; rs1=r1
  
```

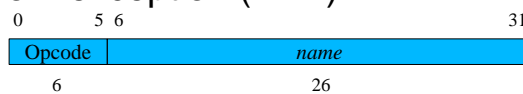
R-type Instructions

- Used for register-to-register ALU ops, read and writes to and from special registers (*IAR* and *FPSR*), and moves between the GPR and/or FPR



J-type Instructions

- Include jump (J), jump & link (JAL), TRAP, and return from exception (RFE)



- name**: 26-bit signed offset that is added to the address of the instruction in the delay-slot (PC+4) to generate the target address
- For TRAP, it specifies an unsigned 26-bit absolute address

j target ; PC=PC+sigext(target)

Load & Store Instructions

- Two categories
 - Load/store GPR
 - Load/store FPR
- All of these are in I-type format

`effective_address = (rs) + sigext(immediate)`

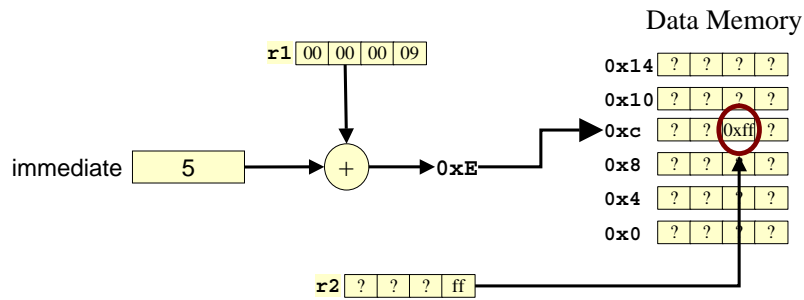
Load & Store GPR

- LB, LBU, SB
- LH, LHU, SH
- LW, SW

`LB/LBU/LH/LHU/LW rd,immediate(rs1)`
`SB/SH/SW immediate(rs1),rd`

Store Byte (Example)

```
; Let r1=9, r2=0xff
sb 5(r1),r2
```

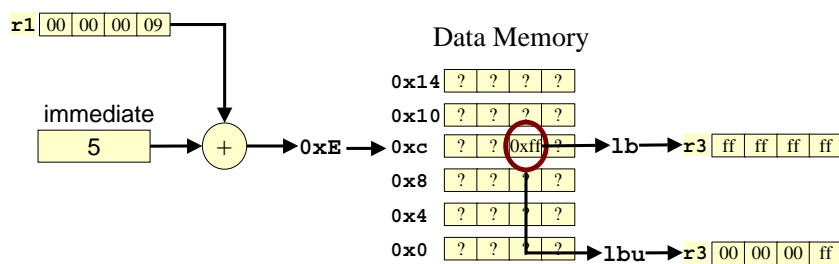


Maurizio Palesi

21

Load Byte (Example)

```
; Let r1=9
lb r3,5(r1)
```



Maurizio Palesi

22

Move Instructions

- All of these are in the R-type format
 - MOVI2S, MOVS2I: GPR ↔ IAR
 - ✓ `movi2s rd,rs1 ; rd∈SR, rs1∈GPR`
 - ✓ `movs2i rd,rs1 ; rd∈GPR, rs1∈SR`
 - MOVF, MOVD: FPR ↔ FPR
 - ✓ `movf rd,rs1 ; rd,rs1∈FPR`
 - ✓ `movd rd,rs1 ; rd,rs1∈FPR even-numbered`
 - MOVFP2I, MOVI2FP: GPR ↔ FPR
 - ✓ `movfp2i rd,rs1 ;rd∈GPR, rs1∈FPR`
 - ✓ `movi2fp rd,rs1 ;rd∈FPR, rs1∈GPR`

Arithmetic and Logical Instructions

- Four categories
 - Arithmetic
 - Logical
 - Shift
 - Set-on-comparison
- Operates on signed/unsigned stored in GPR and Immediate (except LHI that works only by imm)
 - R-type & I-type format
- MUL & DIV works only with FPR

Arithmetic and Logical Instructions

Arithmetic Instructions

- ADD, SUB (`add r1,r2,r3`)
 - Treat the contents of the source registers as signed
 - Overflow exception
- ADDU, SUBU (`addu r1,r2,r3`)
 - Treat the contents of the source registers as unsigned
- ADDI, SUBI, ADDUI, SUBUI (`addi r1,r2,#17`)
 - As before but with immediate operand
- MULT, MULTU, DIV, DIVU (`mult f1,f2,f3`)
 - Only FPR
 - Require MOVI2FP and MOVFP2I

Arithmetic and Logical Instructions

Logical Instructions

- AND, OR, XOR (`and r1,r2,r3`)
 - Bitwise logical operations on the contents of two regs
- ANDI, ORI, XORI (`andi r1,r2,#16`)
 - Bitwise logical operations on the contents of a GPR's regs and the 16-bit *immediate* zero-extended
- LHI (Load High Immediate) (`lhi r1,0xffff00`)
 - Places 16-bit immediate into the most significant portion of the destination reg and fills the remaining portion with '0's
 - Makes it possible to create a full 32-bit constant in a GPR reg in two instructions (LHI followed by an ADDI)

Arithmetic and Logical Instructions

Shift Instructions

- SLL, SRL, SRA (`sll r1,r2,r3`)
 - Shift amount specified by the value of the contents of a GP-reg
- SLLI, SRLI, SRAI (`slli r1,r2,#3`)
 - Shift amount specified by the value of the *immediate* field
- At any rate, only the five low-order bits are considered

Arithmetic and Logical Instructions

Set-On-Comparison Instructions

- SLT, SGT, SLE, SGE, SEQ, SNE

```
sllt r1,r2,r3      ; r1=(r2<r3)? 1:0
sllle r1,r2,r3     ; r1=(r2<=r3)?1:0
sllseq r1,r2,r3    ; r1=(r2==r3)?1:0
```

set the destination register to a value of 1 when the comparison result is 'true' and set the destination register to a value of 0 when the comparison result is 'false'

- SLTI, SGTI, SLEI, SGEI, SEQI, SNEI

```
sllgei r1,r2,#5      ; r1=(r2 >= 5)?1:0
```

as before but with immediate argument (immediate is sign-extended)

Floating-Point Instructions

- Three categories
 - Arithmetic
 - Conversion
 - Set-on-comparison
- All floating-point instructions operate on FP values stored in either an individual (for single-precision) or an even/odd pair (for double-precision) floating-point register(s)
- All are in R-type format
- IEEE 754 standard (refer to the *ANSI/IEEE Std 754-1985 Standard for binary Floating Point Arithmetic*)

Floating-Point Instructions

Arithmetic & Convert Instructions

- ADDF, SUBF, MULTF, DIVF
 - `addf f0,f1,f2`
- ADDD, SUBD, MULTD, DIVD
 - `addd f0,f2,f4`
- CVTF2D, CVTF2I
 - Convert a float to double and integer (`cvtf2d f0,f2`)
- CVTD2F, CVTD2I
 - Convert a double to float and integer (`cvt2d2i f0,r7`)
- CVTI2F, CVTI2D
 - Convert integer to float and double (`cvti2f r1,f0`)

Floating-Point Instructions

Set-On-Comparison Instructions

- LTF, LTD Less Than Float/Double
`l t f f0, f1 ; FPSR=(f0<f1)?true:false`
- GTF, GTD Greater Than Float/Double
- LEF, LED Less Than or Equal To Float/Double
- GEF, GED Greater Than or Equal To Float/Double
- EQF, EQD Equal To Float/Double
- NEF, NED Not Equal To Float/Double

Jump and Branch Instructions

- BEQZ, BNEQ, BFPT, BFPF (I-type)

```
beqz r1,target ; if (r1==0) PC=PC+4+sigext(target)
bnez r1,target ; if (r1==1) PC=PC+4+sigext(target)
bfpt label ; if (fpsr==true) PC=PC+4+sigext(label)
bfpf label ; if (fpsr==false) PC=PC+4+sigext(label)
```

- The branch target address is computed by sign-extending the 16-bit name and adding to the PC+4

Jump and Branch Instructions

■ J, JR, JAL, JALR

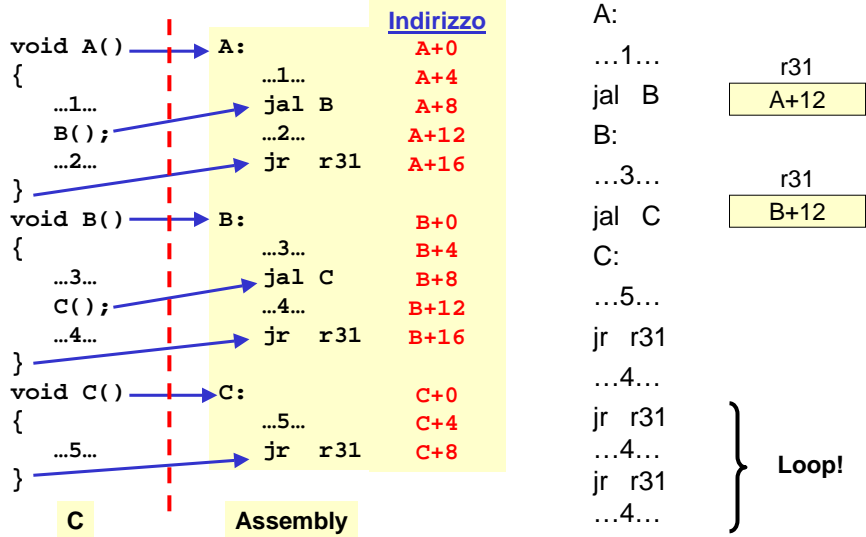
- The target addr of J & JAL is computed by sign-extending 26-bit name field and adding to PC+4
- The target addr of JR & JALR may be obtained from the 32-bit unsigned contents of any GPreg
- JAL & JALR place the address of the instruction after the delay slot into R31

```
j   target           ; PC=PC+4+sigext(target)
jr  r1               ; PC=r1
jal label            ; r31=PC+4; PC=PC+4+sigext(label)
jal r1               ; r31=PC+4; PC=r1
```

Procedure Call

- Procedure call can be obtained using **jal** instruction
 - **jal procedure_address**
 - It sets the **r31** to the address of the instruction following the **jal** (return address) and set the PC to the **procedure_address**
- Return from a procedure can be obtained using the **jr** instruction
 - **jr r31**
 - It jumps to the address contained in **r31**

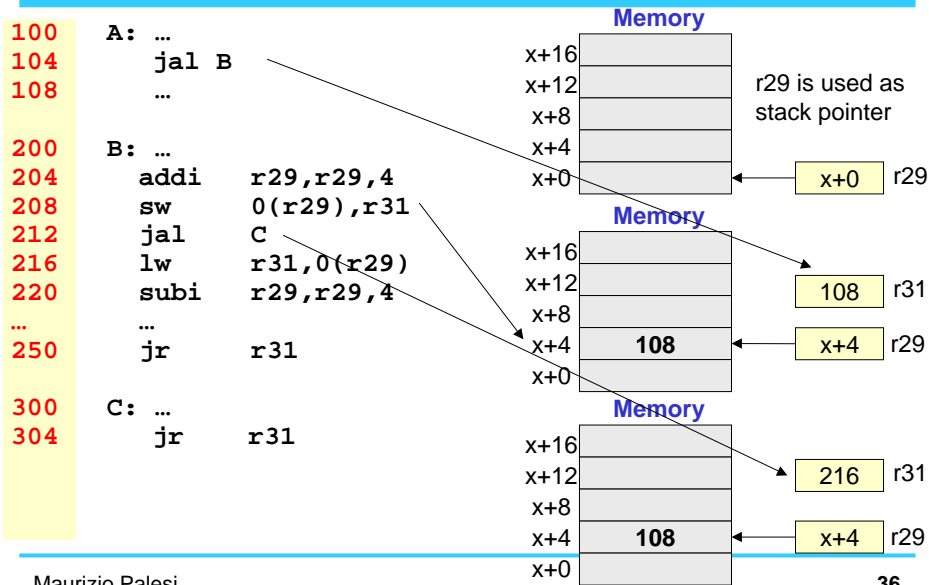
Procedure Call - Lose the Return Address



Maurizio Palesi

35

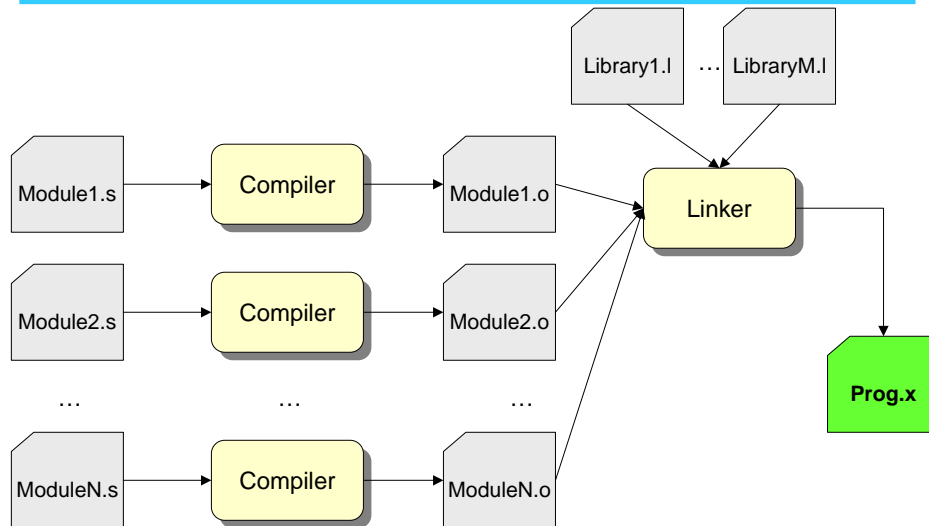
Procedure Call - Using the Stack



Maurizio Palesi

36

Compiler & Linker



Maurizio Palesi

37

Compiler

- Two steps
 - ➔ Building of the symbol table
 - ➔ Substitution of the symbols with values
 - ➔ Language specific: operative code, registers, etc.
 - ➔ User defined: labels, constants, etc.

Maurizio Palesi

38

Unresolved References

■ Why 2 steps?

→ To resolve forward references

✓ i.e., Using a label before its definition

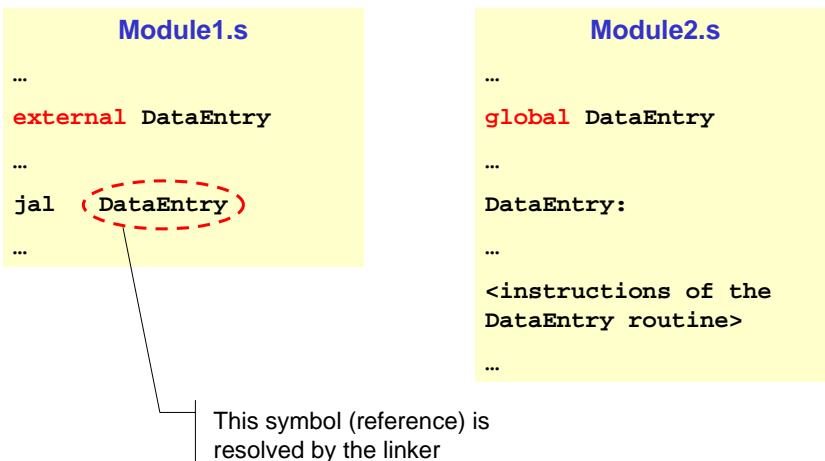
```
bnez(error)  
...  
error:  
...
```

This label has not been defined yet

■ The output file produced by the compiler, namely **object file**, may contains unresolved references to label defined in external files

→ All these references are resolved by the Linker

Local vs Global References



The Object File

- Contains all the information needed by the linker to make the executable file
 - **Header**: size and position of the different sections
 - **Text segment**: binary code of the program (may contains unresolved references)
 - **Data segment**: program data (may contains unresolved references)
 - **Relocation**: list of instructions and data depending on absolute addresses
 - **Symbol Table**: List of symbol/value and unresolved references

Directives

- Assembler directives start with a point (.)
- **.data [ind]**
 - Everything after this directive is allocated on data segment
 - Address **ind** is optional. If **ind** is defined data segment starts from address **ind**
- **.text [ind]**
 - Everything after this directive is allocated on text segment
 - Address **ind** is optional. If **ind** is defined text segment starts from address **ind**

Directives (cnt'd)

- **.word w_1, w_2, \dots, w_N**

→ The 32-bit values w_1, w_2, \dots, w_N are memory stored in sequential addresses

```
.data 100
.word 0x12345678, 0xaabbccdd
```

12	100
34	101
56	102
78	103
aa	104
bb	105
cc	106
dd	107

- **.half h_1, h_2, \dots, h_N**

→ The 16-bit values h_1, h_2, \dots, h_N are memory stored in sequential addresses

- **.byte b_1, b_2, \dots, b_N**

→ The 8-bit values b_1, b_2, \dots, b_N are memory stored in sequential addresses

- **.float f_1, f_2, \dots, f_N**

→ The 32-bit values, in SPFP, f_1, f_2, \dots, f_N are memory stored in sequential addresses

- **.double d_1, d_2, \dots, d_N**

→ The 64-bit values, in DPFP, d_1, d_2, \dots, d_N are memory stored in sequential addresses

Directives (cnt'd)

- **.align <n>**

→ Subsequent defined data are allocated starting from an address multiple of 2^n

```
.data 100
.byte 0xff
.aling 2
.word 0xaabbccdd
```

ff	100
?	101
?	102
?	103
aa	104
bb	105
cc	106
dd	107

- **.ascii <str>**

→ String **str** is stored in memory

```
.data 100
.ascii "Hello!"
```

'H'	100
'e'	101
'\0'	102
'l'	103
'o'	104
'!'	105
'\0'	106
'\0'	107

Directives (cnt'd)

■ **.ascii <str>**

→ String **str** is stored in memory and the byte 0 (string terminator) is automatically inserted

```
.data 100
.ascii "Hello!"
```

'H'	100
'e'	101
'l'	102
'l'	103
'o'	104
'!'	105
0	106

■ **.space <n>**

→ Reservation of **n** byte of memory without initialization

```
.data 100
.space 5
.byte 0xff
```

?	100
?	101
?	102
?	103
?	104
ff	105

■ **.global <label>**

→ Make label be accessible from external modules

Traps - The System Interface (1 of 2)

■ Traps build the interface between DLX programs and I/O-system.

■ There are five traps defined in WinDLX

■ The Traps:

→ **Trap #0**: Terminate a Program

→ **Trap #1**: Open File

→ **Trap #2**: Close File

→ **Trap #3**: Read Block From File

→ **Trap #4**: Write Block to File

→ **Trap #5**: Formatted Output to Standard-Output

Traps - The System Interface (2 of 2)

- For all five defined traps:
 - ➔ They match the UNIX/DOS-System calls resp. C-library-functions `open()`, `close()`, `read()`, `write()` and `printf()`
 - ➔ The file descriptors 0,1 and 2 are reserved for `stdin`, `stdout` and `stderr`
 - ➔ The address of the required parameters for the system calls must be loaded in register R14
 - ➔ All parameters have to be 32 bits long (DPFP are 64 bits long)
 - ➔ The result is returned in R1

Trap #5 Formatted Output to Standard Out

- Parameters
 - ➔ Format string: see C-function `printf()`
 - ➔ ...Arguments: according to format string
- The number of bytes transferred to `stdout` is returned in R1

```
.data
msg:
    .ascii "Hello World!\nreal:%f, integer:%d\n"
    .align 2
msg_addr:
    .word msg
    .double 1.23456
    .word 123456

.text
    addi r14,r0,msg_addr
    trap 5

    trap 0
```

Trap #3

Read Block From File

- A file block or a line from stdin can be read with this trap
- Parameters
 - ➔ File descriptor of the file
 - ➔ Address, for the destination of the read operation
 - ➔ Size of block (bytes) to be read
- The number of bytes read is returned in R1

```
.data
buffer:  .space 64
par:     .word 0
        .word buffer
        .word 64

.text
addi    r14,r0,par
trap    3

trap    0
```

Example

Input Unsigned (C code)

- Read a string from stdin and converts it in decimal

```
int InputUnsigned(char *PrintfPar)
{
    char ReadPar[80];
    int i, n;
    char c;

    printf("%s", PrintfPar);
    scanf("%s", ReadPar);

    i = 0;
    n = 0;
    while (ReadPar[i] != '\n') {
        c = ReadPar[i] - 48;
        n = (n * 10) + c;
        i++;
    }
    return n;
}
```

Example Input Unsigned (DLX-Assembly code)

- Read a string from stdin and converts it in decimal

```
;expect the address of a zero-terminated  
;prompt string in R1 returns the read value in R1  
;changes the contents of registers R1,R13,R14  
  
    .data  
  
    *** Data for Read-Trap  
ReadBuffer: .space 80  
ReadPar:    .word 0,ReadBuffer,80  
  
    *** Data for Printf-Trap  
PrintfPar: .space 4  
  
SaveR2:    .space 4  
SaveR3:    .space 4  
SaveR4:    .space 4  
SaveR5:    .space 4
```

Maurizio Palesi

51

Example Input Unsigned (DLX-Assembly code)

```
.text  
.global InputUnsigned  
  
InputUnsigned:  
    *** save register contents  
sw SaveR2,r2  
sw SaveR3,r3  
sw SaveR4,r4  
sw SaveR5,r5  
  
    *** Prompt  
sw PrintfPar,r1  
addi r14,r0,PrintfPar  
trap 5  
  
    *** call Trap-3 to read line  
addi r14,r0,ReadPar  
trap 3  
  
    *** determine value  
addi r2,r0,ReadBuffer  
addi r1,r0,0  
addi r4,r0,10 ;Dec system  
  
Loop:  
    *** reads digits to end of line  
lbu r3,0(r2)  
seqi r5,r3,10 ;LF -> Exit  
bnez r5,Finish  
subi r3,r3,48 ;'0'  
multu r1,r1,r4 ;Shift decimal  
add r1,r1,r3  
addi r2,r2,1 ;inc pointer  
j Loop  
  
Finish:  
    *** restore old regs contents  
lw r2,SaveR2  
lw r3,SaveR3  
lw r4,SaveR4  
lw r5,SaveR5  
jr r31 ; Return
```

Maurizio Palesi

52

Example Factorial (C code)

- Compute the factorial of a number

```
void main(void)
{
    int i, n;
    double fact = 1.0;

    n = InputUnsigned("A value >1: ");

    for (i=n; i>1; i--)
        fact = fact * i;

    printf("Factorial = %g\n\n", fact);
}
```

Example Factorial (DLX-Assembly code)

```
; requires module INPUT
; read a number from stdin and
; calculate the factorial
; the result is written to stdout

.data
Prompt:
    .asciiz "A value >1: "

PrintfFormat:
    .asciiz "Factorial = %g\n\n"
    .align 2
PrintfPar:
    .word  PrintfFormat
PrintfValue:
    .space 8

.text
.global main
main:
    *** Read from stdin into R1
    addi  r1,r0,Prompt
    jal   InputUnsigned
```

```
*** init values
movi2fp f10,r1
cvti2d  f0,f10    ;D0..Count register
addi    r2,r0,1
movi2fp f11,r2
cvti2d  f2,f11    ;D2..result
movd    f4,f2     ;D4..Constant 1

Loop: *** Break loop if D0 = 1
    led  f0,f4 ;D0<=1 ?
    bfpt Finish

*** Multiplication and next loop
    multd f2,f2,f0
    subd  f0,f0,f4
    j     Loop

Finish: *** write result to stdout
    sd    PrintfValue,f2
    addi  r14,r0,PrintfPar
    trap  5

    trap  0
```

Example ArraySum (C code)

- Compute the sum of the elements of an array

```
#define N 5

void main(void)
{
    int vec[N];
    int i, sum = 0;

    for (i=0; i<N; i++)
        vec[i] = InputUnsigned("A value >1: ");

    for (i=0; i<N; i++)
        sum += vec[i];

    printf("Sum = %d\n", sum);
}
```

Example ArraySum (DLX-Assembly code)

- Compute the sum of the elements of an array

```
.data
vec:      .space 5*4           ; 5 elements of 4 bytes
msg_ins:  .ascii "A value >1: "
msg_sum:  .ascii "Sum: %d\n"
          .align 2
msg_sum_addr: .word msg_sum
sum:      .space 4           ; buffer to store the result

.text
.global main
main:     addi r3,r0,5         ; r3 = N
          addi r2,r0,0         ; r2 = i
data_entry_loop:
          addi r1,r0,msg_ins
          jal  InputUnsigned
          sw   vec(r2),r1
          addi r2,r2,4
          subi r3,r3,1
          bnez r3,data_entry_loop
```

Example ArraySum (DLX-Assembly code)

```
computation:
    addi    r3,r0,5        ; r3 = N
    addi    r2,r0,0        ; r2 = i
    addi    r4,r0,0        ; r4 = sum

loop_sum:
    lw      r5,vec(r2)
    subi    r3,r3,1
    add     r4,r4,r5
    addi    r2,r2,4
    bnez    r3,loop_sum

print:
    sw      sum(r0),r4
    addi    r14,r0,msg_sum_addr
    trap    5

end:
    trap    0
```