Chapter 5

DLX Architecture — A Model RISC Processor

בפרק 4 ראינו תוצאות מחקר על מחשבי CISC ובחרהו מרכיבים ל- ISA בסוג RISC. בפרק זה נציג מודל למעבד RISC בשם DLX. המודל הזה מבוסס על מעבד מסחרי בשם MIPS 2000 ומפושט כדי להקל על לימדת הנושא. לא קיים מעבד DLX בסיליקון ולא כל הפרטים סגורים עד הסוף. לפירוט נוסף ניתן ללמוד ממעבדי MIPS.

Slides 2 **DLX Model Processor**

The DLX is a simplified version of the commercial MIPS 32-bit RISC ISA. The MIPS processor ISA is generally licensed to manufacturers of original equipment (OEMs) who implement the MIPS design in their own products. The DLX is a simplified version of the MIPS ISA, with fewer instructions and a simpler implementation. The DLX is appropriate for learning RISC hardware.

Slides 3 – 4 **DLX Architecture — General Features**

The DLX uses a flat memory model (no segmentation) with a 32-bit address. The ISA defines 3 data types:

Integers (32-bit)

Single precision Floating Point (32-bit)

Double precision Floating Point (64 bits)

The CPU uses a register-register operation model — all ALU operands have source and destination in the registers.

There are 32 integer registers, each 32 bits wide. Integer registers are named R0, R1, ... , R31 and are addressed as 00000 to 11111 in the register address space. Reading register R0 always returns zero — Reg[R0] = 0 (constant). Writes to R0 are legal but have no effect. All other registers are identical (no special purpose registers).

There are 32 FP registers, each 32 bits wide. FP registers are named F0, F1, ... , F31. The DLX uses the IEEE 754 standard FP format. Accesses to double precision FP values use a register pair (even and odd).

The DLX uses three basic addressing modes — register, immediate and memory. The memory addressing mode (**displacement**) can implement **register deferred** (by specifying a zero displacement) and **absolute** (by using R0 that always returns 0).

Slides 5 – 8 **Instruction Set**

The complete instruction set is shown on slides 5 to 9.

Transfer instructions are shown on slide 5. A typical instruction is of the form

`LW R1, 30(R2)`

read as **Load Word** from the memory address **`30(R2)`** to register **`R1`**. The formal definition is

$$\mathbf{Reg[R1] \leftarrow_{32} Mem[30 + Reg[R2]]}$$

which indicates a 32-bit (4-byte) transfer from the memory address formed by adding 30 to the pointer value stored in R2 (and 3 additional bytes from the next 3 addresses).

The **Store Word** instruction has a similar syntax for the opposite direction.

The **Load Byte** instruction transfers one byte from the single memory location and performs a sign extension to a 32-bit representation. Bits in DLX are numbered from 0 = most significant bit (MSB), so that $\mathbf{Mem[30 + Reg[R2]]_0}$ refers to the high-order bit in the byte stored at that address (the sign bit). The notation $\mathbf{(byte_0)^{24}}$ **## byte** means that the 32-bit value is formed from 24 copies of the sign bit followed by the 8 bits of the memory byte.

Notice that **Store Byte** copies the 8 low-order bits from the register to a memory location and that **Load Byte Unsigned** sign extends using leading 0 instead of the sign bit.

The instruction LF is similar to LW but for FP values and registers. The instructions MOVFP2I and MOVI2FP transfer values between integer and FP registers but do not convert any values.

The standard **integer ALU instructions** are defined on slide 6. These include the four basic arithmetic instructions +, –, ×, ÷, the Boolean operations AND, OR, XOR, and six compare operations. The operations ADD, SUB, AND, OR, XOR have register-register versions and versions that operate on a register value and an immediate value. There is also a Load High instruction that writes a 16-bit immediate value to the upper 16-bits of a register. This permits putting the value 0x11223344 into register R2 with the sequence

```
LHI R2, #1122          ; R2 <– 11220000
ORI R2, R2, #3344      ; R2 <– 11220000 OR 00003344 = 11223344
```

The **standard FPU instructions** are defined on slide 7. These include the four basic arithmetic instructions +, –, ×, ÷, and six compare operations for single and double precision. The IEEE 754 standard is presented in chapter 7.

The **control (branch) instructions** are defined on slide 8. The first four instructions are unconditional branches:

J performs an absolute jump.

JAL write the PC to R31 before the jump.

JR jumps to the address stored in a register. R31 can then be used as a return instruction.

JALR is similar to **JAL** but permits a choice of storage register.

There are two conditional branch instructions:

BEQZ performs a conditional jump if the value of a register is 0.

BNEZ performs a conditional jump if the value of a register is not 0.

The **TRAP** instruction performs a software interrupt. The details are not worked out for the DLX but the instruction is well-defined for the MIPS family.

To get a feel for programming in the DLX assembly language we consider an implementation of a simple C program:

```
for ( i = 0 ; i < 256 ; i++)
   a[i] = a[i] + b[i] – c[i] + d[i]
}
```

The arrays contain 256 elements = 256 × 4 = 1024 bytes = 0x400 bytes (hexadecimal notation).

For this example we suppose that the compiler places the 4 arrays at addresses:

```
a[] = 000 – 3FF
b[] = 400 – 7FF
c[] = 800 – BFF
d[] = C00 – FFF
```

The assembly language code is:

```
ADDI R1, R0, #0x400     ; Since R0 = 0 this writes the immediate value 400 to R1
LW   R2, -4(R1)         ; load word from a[] to R2  ( 400 – 4 = 3FC = top of a[] )
LW   R3, 3FC(R1)        ; load word from b[] to R3  (400 + 3FC = 7FC)
ADD  R4, R2, R3         ; add the values in R2 and R3 to R4
LW   R2, 7FC(R1)        ; load word from c[] to R2  (400 + 7FC = BFC)
SUB  R4, R4, R2         ; subtract the value of R2 from R4
LW   R2, BFC(R1)        ; load word from d[] to R2  (400 + BFC = FFC)
ADD  R4, R4, R2         ; add the value of R2 to R4
SW   -4(R1), R4         ; store sum from R4 to a[]
SUBI R1, R1, #4         ; i--  (the index i points to 4-byte memory locations)
BNEZ R1, -0x28          ; if R1 <> 0 jump 10 back instructions
```

Notice that the assembly code works from the top of the array to the bottom, an optimized implementation of the C program that reduces branch instructions and ALU operations.

The BNEZ instruction jumps back in the loop as long as R1 does not reach 0. While BNEZ executes the PC already points to the next instruction below it. An infinite loop is written as

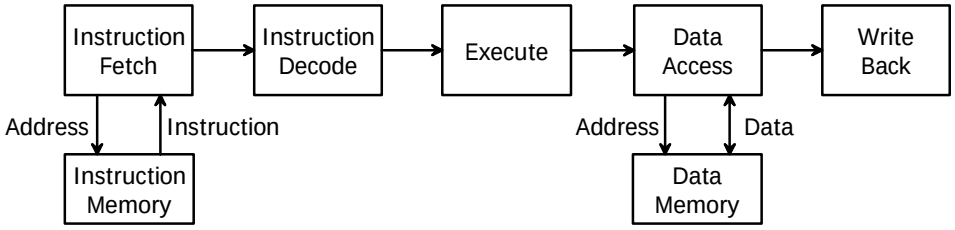
```
J –0x04         ; subtract 4 from the address of the instruction below J.
```

The length of the jump is 10 instructions and all instructions are 4 bytes (32-bits) long. The number 40 is hex is 0x28.

Slide 10 Implementation

The general approach to implementation is division into 5 independent stages. This hardware organization is based on an industrial assembly line with uniform operations. There is no central system bus. There are separate memory caches for instructions and data.

High level design



Instructions move through 5 stages from left to right. The instruction completes in each stage after 1 clock cycle. The first two stages **FETCH** and **DECODE** are identical for all instructions. The last three stages operate differently according to the instruction:

EXECUTE performs ALU instructions and address calculations

MEMORY ACCESS executes Load and Store instructions

WRITE BACK performs register updates for Load and ALU instructions

Slide 11 RISC Performance

Analysis of x86 and SPEC programs on the VAX and MIPS 2000 CPU lead to a significant **speedup**:

$$S = \frac{T}{T'} = \frac{CPI^{CISC} \times IC^{CISC} \times \tau^{CISC}}{CPI^{RISC} \times IC^{RISC} \times \tau^{RISC}} = \frac{CPI^{CISC}}{CPI^{RISC}} \times \frac{IC^{CISC}}{IC^{RISC}} \times \frac{\tau^{CISC}}{\tau^{RISC}} = (\approx 6) \times \left(\approx \frac{1}{2}\right) \times \frac{\tau^{CISC}}{\tau^{RISC}} \approx 3 \times \frac{\tau^{CISC}}{\tau^{RISC}}$$

On average the compilation of a C program to RISC is about twice as large (IC) as for CISC. But the CISC CPU requires about 6 times as many clock cycles to execute a typical instruction as a RISC CPU, so the speedup is at least 3, even before raising the clock rate.

Slides 12 – 15 Instruction Formats

All DLX instructions are 32-bits with bits numbered 0 (left) to 31 (right). There are three instruction formats:

Type	0-5	6-10	11-15	16-31	
	6	5	5	5	11
R	opcode	rs1	rs2	rd	function
I	opcode	rs	rd	immediate	
J	opcode	offset			

In all instructions bits 0 – 5 specify the opcode (operation code) that determines the instruction.

The **J-type** specifies the unconditional branch instructions Jump and Jump And Link. The branch offset field is 26 bits allowing long jumps.

The **R-type** specifies register-register ALU instructions with two source registers (rs1, rs2), a destination register (rd) and a function field that expands the ALU types. The typical operation is of the form **`rd ← rs1 function rs2`**.

The **I-type** is used for all other instructions with one source register (rs), destination register (rd) and a 16-bit immediate value. The typical instructions are:

Loads

```
rd ← imm(rs)
```

Stores

```
imm(rs) ← rd
```

ALU operations with immediate operand

```
rd ← rs op immediate
```

Conditional branch instructions

```
if rs eq/ne 0 then PC ← PC + imm  (rd unused)
```

Jump register

```
PC ← rs
```

Jump and link register

```
rd ← PC
PC ← PC + immediate
```

Implementation Details

We first describe the implementation of instructions in the 5 stages of the CPU and then give detailed examples of execution in the detailed subsystems of the CPU microarchitecture.

Slide 17 **Execution Stages by Instruction Type**

The actions of each instruction type in each CPU stage are shown in the table:

Stage	ALU	Store	Load	Branch
Instruction Fetch (**IF**)	Fetch instruction from memory	Fetch instruction from memory	Fetch instruction from memory	Fetch instruction from memory
Instruction Decode (**ID**)	Decode operation and operands	Decode operation and operands	Decode operation and operands	Decode operation and operands
Execute (**EX**)	Calculate ALU operation	Calculate memory address	Calculate memory address	Calculate branch condition Calculate branch address
Memory (**MEM**)		Update PC	Load data from memory	Update PC
Writeback (**WB**)	Write result to register Update PC		Write loaded data to register Update PC	

We see that all instructions undergo the same actions in the IF and ID stages.

The EX stage uses the ALU in different ways depending on the instruction — it may perform basic ALU operations or calculate address locations for memory and branch instructions. The MEM stage performs memory access for load and store operations. The WB stage updates registers are ALU and load operations.

Slide 18 Temporary Registers for Implementation

In order to carry out instruction execution certain temporary registers are required:

IR — Instruction Register

Register in the IF stage that holds the fetched instruction during execution

PC — Program Counter

Register in the IF stage that holds the memory address of the next instruction

NPC — Next Program Counter

Register in the IF stage that holds a temporary update of the PC

A, B, I

Registers in the ID stage that buffer operand for use in the ALU

ALU_{out} — ALU output

Register in the EX stage that holds the result of an ALU operation

Cond — Condition Flag

Register in the EX stage that holds the result of the test for conditional branch

LMD — Load Memory Data

Register in the MEM stage that holds data loaded from memory

Slide 19 Example Type-I ALU Instruction

We begin with a simple type-I ALU instruction. The analysis of the other instructions types will follow this pattern. Line 1 presents the **assembly syntax** for the instruction, line shows 2 the **formal description** of the operation and line 3 shows the **instruction encoding**. The encoding shows that the opcode is addi, rs = R2 (00010 binary), rd = R1 and the immediate = 5.

1	Instruction	`addi R1, R2, #5`
2	Operation	`Reg[R1] ← Reg[R2] + 5`
3	Encoding	0-5: addi (op) | 6-10: 00010 (rs) | 11-15: 00001 (rd) | 16-31: 0000 0000 0000 0101 (immediate)
4	Hardware Stage 1	`IR ← Mem[PC]` `NPC ← PC + 4`
5	Hardware Stage 2	$A \leftarrow Reg[IR_{6\text{-}10}]$ /* A ← Reg[R2] */ $B \leftarrow Reg[IR_{11\text{-}15}]$ /* B ← Reg[R1] */ $I \leftarrow (IR_{16})^{16}$ ## $IR_{16\text{-}31}$
6	Hardware Stage 3	$ALU_{out} \leftarrow A + I$
7	Hardware Stage 4	
8	Hardware Stage 5	$Reg[IR_{11\text{-}15}] \leftarrow ALU_{out}$ /* Reg[R1] ← A + I */ `PC ← NPC`

The execution lines of the table are:

4	Hardware Stage 1	$IR \leftarrow Mem[PC]$ $NPC \leftarrow PC + 4$
5	Hardware Stage 2	$A \leftarrow Reg[IR_{6-10}]$ /* A ← Reg[R2] */ $B \leftarrow Reg[IR_{11-15}]$ /* B ← Reg[R1] */ $I \leftarrow (IR_{16})^{16}$ ## IR_{16-31}
6	Hardware Stage 3	$ALU_{out} \leftarrow A + I$
7	Hardware Stage 4	
8	Hardware Stage 5	$Reg[IR_{11-15}] \leftarrow ALU_{out}$ /* Reg[R1] ← A + I */ $PC \leftarrow NPC$

Line 4 describes two actions in stage 1 — the **instruction fetch** stage:

The IF stage reads the address of the next instruction from PC, accesses this address in the instruction cache and stores the 4-byte instruction in IR.

The IF stage adds 4 to the PC and stores the result in NPC.

Line 5 describes three actions in stage 2 — the **instruction decode** stage:

The ID stage reads the 5-bit number of rs from the instruction, accesses this register and copies the 4-byte value to A.

The ID stage reads the 5-bit number of rd from the instruction, accesses this register and copies the 4-byte value to B.

The ID stage reads the 16-bit immediate from the instruction, sign-extends to 32 bits and copies the 4-byte value to I.

Line 6 describes the action in stage 3 — the **execution stage**:

The EX stage adds the values of A and I and writes the 4-byte value to ALU_{out}. Notice that the value in B is ignored.

Line 7 shows that no action is taken in stage 4 — the **memory** stage. This stage can be skipped by instructions of this type.

Line 8 describes the two actions in stage 5 — the **writeback** stage:

The WB stage reads the 5-bit number of rd from the instruction, and writes the value of ALU_{out} to this register.

The WB stage copies the address of the next instruction from NPC to PC. This completes the instruction execution and prepares for the fetch of the next instruction.

Slide 20 Example Type-R ALU Instruction

This instruction is similar to the type-I example except that is uses R3 instead of an immediate. The execution stages are also identical, except in the execution stage:

The EX stage adds the values of A and B and writes the 4-byte value to ALU_{out}. This time the value in I is ignored.

Type-I ALU uses **I** and type-R ALU uses **B**. It is simpler to prepare both **I** and **B** in stage ID than to add extra hardware that makes the choice. The extra register read is unnecessary but harmless.

Slide 21 **Example Type-I Store Instruction**

This instruction is similar to the type-I ALU instruction except that the sum is used as a memory address instead of being written to a register. The actions in stages 1 – 3 are identical to those of the type-I ALU instruction. But here

The MEM stage writes the value of B to memory using the 4-byte address value from ALU_{out}.

The MEM stage copies the address of the next instruction from NPC to PC. This completes the instruction execution and prepares for the fetch of the next instruction.

Slide 22 **Example Type-I Load Instruction**

This instruction is similar to the type-I load instruction except that direction of the transfer is reversed:

The MEM stage reads from memory the 4-byte value at the address from ALU_{out} and copies the operand to LMD.

The WB stage copies the loaded operand from LMD to the register named in the instruction in rd.

The WB stage copies the address of the next instruction from NPC to PC. This completes the instruction execution and prepares for the fetch of the next instruction.

Slide 23 **Example Type-I Conditional Branch Instruction**

The conditional branch operates differently from the previous instructions. As always, the actions in stage IF and ID are the same as for every instruction. Then

The EX stage adds the values of NPC and I and writes the 4-byte value to ALU_{out}. This value is the address of the branch target instruction if the branch condition is valid.

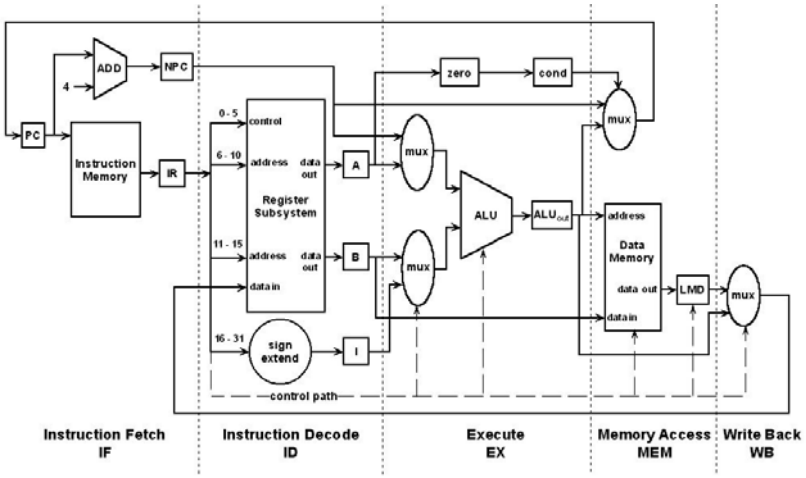
The EX stage compares the value in A with 0 and writes a 1-bit flag to the condition register cond.

The MEM stage updates the PC register. If the condition flag cond is 1 then it copies ALU_{out} to PC (this is the branch taken). If cond = 0 then it copies NPC to PC (this is the branch not taken).

Slides 24 – 45 **DLX Hardware Drawing — Version 1**

The hardware drawing of DLX version 1 should be read as a graphic representation of the formal description given above.

The 5 stages are separated by dotted lines. The solid lines are data paths among basic elements (memory, registers, ALU) and temporary registers — they implement the actions described on slides 19 – 23.



Stages EX, MEM and WB contain **multiplexors** labeled as **mux**. These multiplexors implement the choices described in the description of each stage — the control path (dashed line) provides a signal that chooses one input from the left and sends it to the output at the right.

The EX stage has two multiplexors. The lower multiplexor has inputs I and B — it selects I for type-I instructions and selects B for type-R ALU instructions. The upper multiplexor selects input A for all instruction types except conditional branch when it selects NPC.

The multiplexor in MEM selects NPC, except for a conditional branch taken when it selects the calculated target address from ALU_{out}.

The multiplexor in WB selects ALU_{out}, except for load instructions when it selects LMD.

Slides 25 to 28 show the stage-by-stage execution of the Type-I ALU Instruction.

Slide 25 shows the actions performed in stage 1 — IF and the information flow

Slide 26 shows the actions performed in stage 2 — ID and the information flow

Slide 27 shows the actions performed in stage 3 — EX and the information flow

Slide 28 shows the actions performed in stage 5 — WB and the execution completion

Slides 29 to 32 show the stage-by-stage execution of the Type-R ALU Instruction.

Slides 33 to 36 show the stage-by-stage execution of the Type-I Store Instruction.

Slides 37 to 41 show the stage-by-stage execution of the Type-I Load Instruction.

Slides 42 to 45 show the stage-by-stage execution of the Type-I Branch Instruction.

Slide 46 Performance

As seen in the detailed analysis of instruction execution, all instructions DLX version 1 can execute in 4 clock cycles, except for loads, which require 5 clock cycles.

Hennessey and Patterson report that compilation of SPEC CINT programs for the DLX instruction set can be sorted to produce the following table for IC_i / IC along with CPI_i:

Instruction Type i	IC_i / IC	CPI_i
ALU	40%	4
Load	25%	5
Store	15%	4
Branch	20%	4

The average CPI can be calculated as

$$CPI = \sum_i CPI_i \times \frac{IC_i}{IC} = 4 \times 0.40 + 5 \times 0.25 + 4 \times 0.15 + 4 \times 0.25 = 4.25$$

The goal of future versions of the DLX is to reduce this value of CPI to as close to 1 as possible.