Presentation 2

DLX: A Simplified RISC Model

באמצע שנות ה- 1980 החוקרים John L. Hennessy (סטנפורד) ו- David A. Patterson (ברקלי) הובילו את הפיתוח של גישת RISC בארכיטקטורה. אחד המעבדים הראשונים בגישה הזאת היה מוצר בשם MIPS2000. החוקרים גם שיתפו פעולה בספר Computer Architecture: A Quantitative Approach ולצורך כך כתבו ISA בשם DLX ככלי הוראה מבוסס על משפחת MIPS. המודל מהווה בסיס מסודר ללימוד מושגי RISC, ומתאים להצגת פיתוחים עכשוויים שלא קיימים במודל המקורי. מטרת המצגת היא לתאר את ה- ISA והמימוש הבסיסי של DLX.

Slide 2 – 4 **DLX Pipeline**

The DLX pipeline contains the "classic" 5 stages of first generation RISC: IF, ID, EX, MEM, WB. Instructions are executed in an integer ALU or floating point unit (FPU).

DLX is a 32-bit machine using 32-bit memory addresses and operating on 32-bit integers and floats (single precision).

The CPU contains 32 integer registers R0 – R31 and 32 FP registers F0 – F31. Double precision FP numbers are 64 bits and stored in a pair of FP registers. All integer registers are identical except R0 which is read-only (contains 0). A write to R0 is legal but has no effect.

The blue lines in the pipeline diagram perform **forwarding** (also called bypass) — intermediate results of can be fed backward to instructions that need them to prevent RAW hazards (explained in detail on slides 16 – 29).

Slide 5 **DLX Instruction Formats**

The DLX **instruction format** is very similar to the Alpha instruction (presentation 1 slide 38). Bits are numbered from left to right.

Bits 0 – 5 are the opcode (identify the operation).

For **Type J** instructions (unconditional jump), bits 6 – 31 provide an offset (displacement) to add to PC to implement the branch.

In **Type R** instructions (reg-reg ALU), 3 registers are named in bits, 6 – 10, 11 – 15, and 16 – 20. With 5 bits per register, this method can name 32 registers, 0 – 31. The first registers are sources and the third is the destination.

The **Type I** format is used for instructions requiring an **immediate** literal value: load, store, ALU with immediate, and branch.

Slide 6 – 9 **DLX Instruction Set**

The DLX has a small instruction set, divided into **transfers**, **integer ALU**, **FP**, and **control**. Each instruction is defined in the formal language described in presentation 1. For example, the first instruction is the Load Word instruction:

$$\mathtt{LW\ R1,30(R2)} \qquad \mathtt{R1} \leftarrow_{32} \mathtt{MEM[30+REGS[R2]}$$

The operation adds 30 to the 32-bit value in register **R2**, uses that value as an address in memory, and loads the 32-bit value at that address (4 consecutive bytes) to **R1**. Store Word is

similar but copies the value from register to memory. Load Float loads the value to a floating point register instead of an integer register.

A subscript after a value refers to a single bit from the value (numbered right to left).

A superscript after a value indicates repetition of the bit value.

The symbol ## is concatenation (שרשור).

The instruction Load Byte reads one byte from a memory address, and copies the high order bit (the sign of a signed integer) 24 times to the left of the byte. The result is a 32-bit representation of the signed value of the byte.

For example, the signed value –2 (decimal) = 0xFE (hexadecimal byte) is expanded to the 32-bit value 0xFFFFFFFE representing –2.

The instruction LBU is Load Byte Unsigned — instead of the upper bit, the instruction fills in 24 leading zeros for the unsigned valued.

The instructions MOVFP2I and MOVI2FP transfer literals unchanged between register sets. There is no conversion of float to int or int to float.

The **integer ALU** register-register instructions are ADD, SUB, MULT, DIV, AND, OR, and XOR. The forms ADDI, SUBI, ANDI, ORI, XORI take one immediate source operand. There are 6 comparison instructions that set a register based on the comparison of two other registers.

The **floating point** instructions are ADD, SUB, MULT, DIV, and the 6 comparisons. The register StatFP is a 1-bit flag used to hold the comparison results.

There are 7 **control instructions**:

J offset is an unconditional jump — it adds the offset to the PC.

JAL is similar but it saves the default PC (a return point) in register R31.

The saved return point can be accessed using the **JL reg** instruction that loads PC with the value saved in the register.

JALR reg, offset is similar to JAL but allows the choice of register for saving the return point.

BEQZ reg, offset is **B**ranch on **Eq**ual **Z**ero — it adds the offset to PC is the register contains 0. **BNEZ** is similar but jumps when the register is non-zero. These instructions implement the **conditional branches** used in high level control blocks — if, else, for, while, case, ...

TRAP N is a software interrupt. In some implementations, N is an index to a table of ISR addresses, while in others it is the address of the ISR.

Slide 10 Programming in DLX Assembly

The simple C program

```
main()
{
      int i,j;
      for (i = 0; i < 10; i++){
            j = 2 * i;
      }
}
```

written for DLX appears

```
          ADDI R1, R0, #0        ; i = R1 <-- 0
          ADDI R10, R0, #0A      ; R10 <-- 10
start:    SGE R11, R1, R10       ; R11 <-- 1 iff R1 >= R10 = 10
          BNEZ R11, stop         ; jump to label stop if R1 >= 10
          ADD R2, R1, R1         ; R2 <-- R1 * 2
          ADDI R1, R1, #1        ; R1++
          J start                ; jump to start
stop:     SW -2(R13), R2         ; store j <-- R2
                                 ; R13 = base pointer for variables
          JR R31                 ; return to calling function
```

Slide 11 – 14 DLX Implementation (Integer Pipeline)

The pipeline consists of 5 stages **IF**, **ID**, **EX**, **MEM**, **WB** separated by **4 stage buffers**.

The buffers are named for the stages they separate: **IF/ID** (between IF and ID), **ID/EX**, **EX/MEM**, **MEM/WB**. The stage buffers contain temporary registers that hold intermediate values during instruction execution. The temporary registers are defined on slide 12. It is useful to think of the stage buffer as a **struct** in a C program, with each register as one member. The formal specification on slides 13 – 14 describes the operations in each stage.

Each stage buffer operates as an **edge triggered latch** (דלגלג מדורבן מעבר). The input to the latch does not affect the contents except on a falling transition of the clock signal **CLK**. At the precise moment that CLK goes high-to-low, the flip-flop samples and stores the input, and provides the new stored value as an output. This produces **synchronous transfer** of data (synchronized with the clock signal CLK).

IF stage

On the CLK transition, **PC** updates. The cond flag is only 1 if there is a conditional branch taken (the branch condition is true). If cond = 1 then the next **PC** is the computed address taken from the adder in stage ID (this case is described in more detail below). Otherwise the **PC** receives the default address of the **fall-through instruction** (the next instruction in the listing) found by adding 4 bytes per instruction.

$$\mathbf{PC} \leftarrow \begin{cases} \mathbf{PC + 4,\ cond = 0} \\ \mathbf{ID/EX.NNPC,\ cond = 1} \end{cases}$$

This means that when **PC** updates, it holds its new value for a full clock cycle τ with the value of **PC + 4** waiting as a new input. The new input becomes the updated **PC** on the following CLK transition.

The same value loaded to **PC** is saved in the register **NPC** (next **PC**) in the stage buffer IF/ID.

$$\mathbf{IF/ID.NPC} \leftarrow \begin{cases} \mathbf{PC + 4,\ cond = 0} \\ \mathbf{ID/EX.NNPC,\ cond = 1} \end{cases}$$

The current **PC** (the value stored in the latch) is used as a memory address. The 32-bit instruction at that address is loaded into the instruction register IR.

$$\mathbf{IF/ID.IR} \leftarrow \mathbf{Mem[PC]}$$

ID stage

In the decode stage several temporary registers receive values:

A receives the content of rs1 (type R) or rs (type I):

$$\mathbf{ID/EX.A} \leftarrow \mathbf{Reg[IF/ID.IR_{6\text{-}10}]}$$

B receives the content of rs2 (type R) or rd (type I):

$$\mathbf{ID/EX.B} \leftarrow \mathbf{Reg[IF/ID.IR_{11\text{-}15}]}$$

I receives the last 16 bits in the instruction, extended as a signed number to 32 bits. This is either meaningless (type R) or is the immediate literal in the instruction (type I):

$$\mathbf{ID/EX.I} \leftarrow \mathbf{(IR_{16})^{16}\ \#\#\ IF/ID.IR_{16\text{-}31}}$$

IR receives the instruction encoding:

$$\mathbf{ID/EX.IR} \leftarrow \mathbf{IF/ID.IR}$$

NNPC receives the value NPC + immediate. If there is a taken branch, NNPC holds the value of the computed address for the target:

$$\mathbf{ID/EX.NNPC} \leftarrow \mathbf{IF/ID.NPC + (IR_{16})^{16}\ \#\#\ IF/ID.IR_{16\text{-}31}}$$

cond receives the condition flag, which is 1 if the value in register A is 0:

$$\mathbf{ID/EX.cond} \leftarrow \mathbf{(Reg[IF/ID.IR_{6\text{-}10}] == 0)}$$

EX stage

In EX the ALU receives two operands from two multiplexors:

$$\mathbf{EX/MEM.ALU_{OUT}} \leftarrow \begin{cases} \mathbf{ID/EX.A\ function\ ID/EX.B} & \mathbf{(R\text{-}ALU)} \\ \mathbf{ID/EX.A\ op\ ID/EX.I} & \mathbf{(I\text{-}ALU,\ Memory)} \end{cases}$$

$$\mathbf{Forwarding:\ EX/MEM.ALU_{OUT}\ or\ MEM/WB.ALU_{OUT}\ or\ MEM/WB.LMD}$$
$$\mathbf{substituted\ for\ A\ or\ B}$$

For type R instructions, the top multiplexor provides the value from temporary register **A** and the bottom multiplexor provides the value from temporary register **B**.

For type I instructions, the top multiplexor provides the value from temporary register **A** and the bottom multiplexor provides the value from temporary register **I**.

The forwarding mechanism permits substitutions for either **A** or **B**. This mechanism is explained in slides 21 – 25.

The ALU performs the instruction specified in **ID/EX.IR** and writes the result to $\mathbf{ALU_{out}}$.

Also, the values in registers **B** and **IR** are copied from the stage buffer at left to the stage buffer at right:

$$EX/MEM.B \leftarrow ID/EX.B$$
$$EX/MEM.IR \leftarrow ID/EX.IR$$

MEM stage

On load operations, the value in ALU_{out} is used as the memory read address — the data from memory is written to temporary register **LMD**.

On store operations, the value of register **B** is written to memory at address ALU_{out} . The forwarding mechanism permits substitution of $MEM/WB.ALU_{out}$ for **B**.

$$MEM/WB.LMD \leftarrow Mem[EX/MEM.ALU_{OUT}] \ (Load)$$
$$Mem[EX/MEM.ALU_{OUT}] \leftarrow EX/MEM.B \ (Store)$$
$$Fowarding: MEM/WB.ALU_{OUT} \ substituted \ for \ B$$

For other instruction types, there is no memory access.

The values in ALU_{out} and IR are copied from the stage buffer at left to the stage buffer at right.

$$MEM/WB.ALU_{OUT} \leftarrow EX/MEM.ALU_{OUT}$$
$$MEM/WB.IR \leftarrow EX/MEM.IR$$

WB stage

ALU and load operations write results to registers according to the destination location **rd** (bits 11 – 15 for type I, or bits 16 – 20 for type R).

$$Reg[MEM/WB.IR_{11\text{-}15}] \leftarrow \begin{cases} MEM/WB.ALU_{OUT} \ (ALU\text{-}I) \\ MEM/WB.LMD \ (Load) \end{cases}$$
$$Reg[MEM/WB.IR_{16\text{-}20}] \leftarrow MEM/WB.ALU_{OUT} \ (ALU\text{-}R)$$

General features of the implementation

The operations in stages IF and ID are identical for every instruction, simplifying the pipeline. This method involves some unnecessary work: For type R instructions, writing to register **I** has no meaning. For type I ALU instructions, writing to register **B** in ID is useless (it receives the old value of **rd**). But this work requires no extra processing time and is harmless. It would require more effort to prevent these actions than to ignore them.

The choice of source operands and operation for the ALU in stage EX permits the CPU to perform any defined operation. Comparing the definitions on slide 5, the implementations are:

Type R ALU:
```
rd ← ALU_function (rs1, rs2)
rd ← ALU_function (A, B)
```

Type I ALU:
```
rd ← ALU_operation (rs, imm)
rd ← ALU_operation (A, I)
```

Load:
```
rd ← imm(rs)
rd ← MEM[A+I]
```

Store:
```
imm(rs) ← rd
MEM[A+I] ← B
```

Branch:
```
if (rs == 0) {PC ← PC + imm}
if (A == 0) {PC ← NPC + I}
```

Slide 15 – 19 Implementation Examples

Each example illustrates:

1. The **instruction** format in assembly language.
2. The **operation** performed by the instruction as a formal specification.
3. The **encoding** of the instruction. The opcodes are entered by name — in the actual instruction these are 6-bit binary numbers representing the operation.
4. The operations performed in **Hardware Stage 1**. These are identical for every instruction.
5. The operations performed in **Hardware Stage 2**. These are identical for every instruction. Some operations are not used by stages 3 – 5.
6. The operations performed in **Hardware Stage 3**. These actions depend on the particular instruction.
7. The operations performed in **Hardware Stage 4**. These actions depend on the particular instruction — some instructions do no work in this stage.
8. The operations performed in **Hardware Stage 5**. These actions depend on the particular instruction — some instructions do no work in this stage.

Slide 20 DLX Integer Pipeline Statistics

The **instruction distribution** is found by compiling the programs in SPEC Cint into DLX assembly language and sorting the instructions. The result is

ALU	40%
Load	25%
Store	15%
Branch	20%

Data dependencies between instructions determine RAW hazard statistics. If I_N is an ALU instruction, then in 50% of cases,

One **source** operand of I_N is the **destination** operand of instruction I_{N-1}

The instruction I_{N-1} could be an ALU or load instruction

The DLX must treat RAW hazards for 50% of 40% = 20% of all instructions.

Slide 21 Data Hazards in DLX Integer Pipeline

The DLX must treat RAW hazards for 50% of 40% = 20% of all instructions.

WAW and WAR hazards cannot occur in the DLX

Slide 22 – 23 ALU-ALU RAW Dependencies

The program on slide 22 has 3 data dependencies. Register R1 is the destination of I_1 and a source for $I_2 – I_4$.

I_1 updates R1 in stage WB in CC5. Any instruction that enters stage ID to read R1 before CC5 will read the old value and cause an error.

Unless the hazard is prevented, the table shows that:

I_2 will enter ID in CC3 causing an error

I_3 will enter ID in CC4 causing an error

I_4 enters stage ID in CC5. This does not cause an error, because I_1 in stage WB updates R1 at the **beginning of CC5**, but the reading of R1 for I_4 is not latched into the temporary register **A** or **B** until the next CLK transition at the **end of CC5**. (For those interested, this is shown in detail in slide 23.)

Slide 24 – 25 Pipeline Stall to Avoid RAW Hazard

The hazard can be prevented by stalling the pipeline. I_2 is held in stage IF for 2 clock cycles and enters ID when I_1 performs the WB in CC5. Meanwhile 2 NOP bubbles are placed in the pipeline (as explained in presentation 1 slide 47).

In the instruction view, the red clock cycles indicate the 2 CC pipeline stall, with I_2 held in IF.

The effect on performance is found from

CPI^{stall} = (CC per stall) × (stalls per ALU instruction) × (ALU instructions per instruction)

Since the stall is 2 clock cycles, 50% of ALU instructions (on average) have a data dependency, and 40% of instructions are ALU instructions, CPI^{stall} = 0.40 causing a 29% performance degradation.

Slide 26 – 27 Forwarding or Bypass

Without the pipeline stall, I_2 requires the new value of R1 in stage EX in CC4. During CC4, the new value has not been written to R1, but is **saved** in the stage buffer **`EX/MEM.ALUout`**.

The method called **forwarding** (bypass) is:

1. Allow I_2 to read the old value of R1 in stage ID in CC3 (no pipeline stall).
2. When I_2 reaches stage EX in CC4, do not use the old value. Instead, use the temporary value from **`EX/MEM.ALUout`**.

The green line fed back from EX/MEM to the EX stage represents this forwarding.

Instruction I_3 receives similar treatment. I_3 enters ID in CC4 and reads the old value of R1. When I_3 enters EX in CC6, the temporary value (not yet saved to R1) is saved in stage buffer **`MEM/WB.ALUout`** and can be fed back to the EX stage. The purple line fed back from MEM /WB to the EX stage represents this forwarding.

On the table, the forwarding is indicated by the arrows showing transfer of the temporary result directly to the instruction in stage EX.

In the instruction view, arrows again show transfer of the temporary result directly to the instruction in stage EX.

This way, forwarding removes the need for a pipeline stall.

Slide 28 – 30 Load-ALU RAW Dependencies

The program on slide 28 is similar to the program on slide 22, except that now I_1 is a **load** instruction. The forwarding method can be applied again, but it does not solve the whole problem. The **LW** instruction calculates the memory address in CC3 but only reads from memory in CC4. So, instruct3.ion I_2 must be held in stage ID for 1 extra CC — it enters ID on CC5 and the old value of R1 (that it read in ID) is replaced by the new temporary value saved in register **`MEM/WB.LMD`**.

Comparing the tables on slides 26 and 29,

> The ALU-ALU dependency is handled by copying the value for R1 from EX in CC3 down to EX in CC4.
>
> The Load-ALU dependency is handled by copying the value for R1 from MEM in CC4 down and back to EX in CC5.

The effect on performance is found from

$$CPI^{stall} = (\text{CC per stall}) \times (\text{stalls per ALU instruction}) \times (\text{ALU instructions per instruction})$$

Since the stall is 1 clock cycles, 50% of ALU instructions (on average) have a data dependency, and 25% of instructions are load instructions, $CPI^{stall} = 0.125$ causing an 11% performance degradation. The performance in this situation can be improved in the compiler (slide 35).

Slide 31 ALU-Store RAW Dependencies

The program on slide 30 has another data dependency — the value of R1 is not updated in CC3 when the store instruction SW reads it for writing to memory.

Forwarding can also be used in this case. The temporary result saved in **`MEM/WB.ALU`** replaces the register **B** that holds the old value of R1 read in stage ID. This prevents the hazard without a stall.

Slide 32 – 33 DLX Control Hazard

In order to minimize the control hazard, the DLX uses a policy called **predict not taken**:

1. The branch instruction is evaluated in stage ID. In the same clock cycle, the CPU automatically fetches the default instruction — the next instruction in the program listing (known as the **fall-through** instruction). In the example on slide 31 this occurs on CC2.
2. If the branch instruction evaluates as **not taken** (the condition is false and there is no jump), then the fall-through instruction continues. There is no stall.
3. If the branch instruction evaluates as **taken** (the condition is true and program control jumps), then the fall-through instruction is cancelled and the **target** instruction is loaded from the address calculated from the branch instruction. There is a stall of 1 clock cycle, because the cancelled fall-through moves through the pipeline as a NOP bubble.

The effect on performance is found from

$$CPI^{stall} = (\text{CC per stall}) \times (\text{stalls per ALU instruction}) \times (\text{ALU instructions per instruction})$$

Statistics show that of all branch instructions (20% of all instructions), 2/3 are taken and 1/3 are not taken.

Since the stall is 1 clock cycles, 67% of branch instructions are taken (causing a stall), and 20% of instructions are branch instructions, $CPI^{stall} = 0.13$ causing a 12% performance degradation. The control hazard can be improved by branch prediction (presentation 4).

Slide 34 Other Stalls

Some data dependency stalls are too complex to repair with forwarding:

1. ALU followed by a branch instruction conditioned on the ALU result
2. ALU followed by an independent instruction followed by a store dependent on the ALU result.

In these cases, a stall must occur until the dependent instruction can read the register directly in stage ID while the prior instruction updates the register in WB.

Slide 35 – 36 Rescheduling

Rescheduling is a compiler optimization that can improve performance by preventing hazards.

The program on the left side of slide 35 suffers 3 Load-ALU stalls (1 CC each) and an ALU-Branch stall (2 CC).

Without affecting the program outcome, the compiler can move instructions in the listing so that instruction results are ready when the next dependent instruction needs them. This is called **hiding the latency**.

On the left, the instruction in row 10 is **`SUBI R1, R1, #4`**. In this program R1 is an index for loop iterations and is checked in row 11 (**BNEZ**). On the right, this index is updated in row 2 instead of row 11. Since R1 is also used to index memory accesses (LW and SW), the addresses must be adjusted to account for the change in program order.

On the left, the program performs 2 loads and then an ALU operation. On the right all the loads are performed first, using additional registers for all the loaded data. Now, all the loads are available in registers when the ALU instructions need them. For example, in row 3, LW R2 writes R2 in CC7 and is first used by ADD in row 7 that enters ID in CC8.

The execution of the two programs is shown on slide 36. Each iteration of the loop suffers 5 stall cycles in the original version (plus the control stall after BNEZ). The rescheduled version has no stalls (except the control stall). The loop runs 256 times (100 hexadecimal) and so rescheduling saves 5 × 256 = 1280 clock cycles.

Rescheduling techniques are discussed in detail in presentation 3.

Slide 37 DLX Memory Hierarchy

Slide 37 shows the relationships among memory units in the DLX. The **Instruction Memory** in stage IF and the **Data Memory** in stage MEM of the pipeline are **level 1 (L1) cache**. The L1 cache memories are connected to the cache controller. When a memory address (PC or data address formed in the ALU) is not located in L1 (L1 cache miss), the controller simultaneously accesses level 2 (L2) cache in the DLX package and Main Memory (through the external I/O controller). If the memory location is found in L2 cache (L2 cache hit), the Main Memory access is cancelled. If the location is not found in L2 cache (L2 cache miss), then the location is copied to L2 and to L1. Cache organization is discussed in detail in presentation 4.

Slide 38 – 42 MIPS Architecture

The DLX is a pedagogical abstraction of the commercial **MIPS ISA**. The ISA defines registers and the instruction set for a family of MIPS implementations. Each MIPS core defines the device-dependent implementation details, including pipeline organization, I/O organization, control registers and so on. The **MIPS32** ISA defines a 32-bit RISC CPU similar to the DLX. The **MIPS64** ISA defines a 64-bit RISC CPU. The 32 and 64 bit versions have the same instruction set, with binary compatible machine instructions of 32-bit length. MIPS core designs are typically licensed to OEMs (original equipment manufacturers) that implement the design in an embedded system (a microprocessor based device that is not a general purpose computer).

The register set and instruction format for MIPS is very similar to DLX. An important difference is the set of coprocessors defined for MIPS. For simplicity, certain basic functions expected in a general purpose computer are moved to **coprocessors**, which are not required on simple embedded systems. The ISA defines special instructions that access the coprocessors through the following hardware interfaces:

> CP0 is reserved for virtual memory support and exception handling. It is used to translate virtual addresses (as seen by software) into physical addresses (required by the memory I/O system). It also controls the cache subsystem, handles switches between kernel, supervisor, and user states, and manages exceptions, diagnostic control, and error recovery.
>
> CP1 is used for the interface with an external FPU on older MIPS32 cores.
>
> CP2 is used to interface specialized, device-specific hardware.
>
> CP3 is used for the interface with an external FPU on MIPS644 and newer MIPS32 cores.

Slide 41 shows some of the **MIPS instructions** not defined for the DLX. In addition to the coprocessor instructions, the MIPS defines shift and rotate operations, sync operations to support parallel programming (discussed in presentation 7), predefined systems calls (similar to Alpha PALcode, presentation 1, slide 38), and cache prefetch (discussed in presentation 4). Other instructions include convenient variations, such as Set on Less Than Immediate, Branch on greater or equal zero, and Branch on less than or equal zero.