Chapter 6

Speeding Up DLX

בפרק 5 הצגנו את מעבד ה- DLX כמודל למעבד RISC וביצענו ניתוח של ה- microarchitecture שלו. גרסה 1 של ה- DLX פועלת על פקודה אחת בחמישה שלבי ביצוע ומסיים את הטיפול בפקודה הייצוגית ב- 4.25 מחזורי שעון. בפרק הזה נציג גרסה משופרת שמסוגלת לסיים ביצוע פקודה כמעט על כל מחזור שעון. הכלי הבסיסי לשיפור נקרא pipeline (צנרת) והוא מאפשר חפיפה של פקודות. כאשר פקודה מסיימת בחטיבת IF ועוברת לחטיבת ID פקודה חדשה נכנסת לחטיבת IF וכך הלאה. ברוב המקרים המעבד פועל על 5 פקודות, בשלבי ביצוע שונים, בכל עת. כדי לאפשר לימוד מסודר נכניס שיפור אחד לפי הצורך ואז נדון בשיפור נוסף. כך נעבור מגרסה 1 לגרסה 2, 3, 4, 5, 6A, 6B, 6C.

Slides 2 – 3 **DLX Execution Stages — Version 1**

Slide 2 reviews the executions stages of DLXv1:

Clock Cycle 1

I_1 (instruction number 1) enters Instruction Fetch (IF)

Clock Cycle2

I_1 moves to Instruction Decode (ID)
Instruction Fetch (IF) holds its state fixed

Clock Cycle3

I_1 moves to Execute (EX)
Instruction Fetch (IF) holds its state fixed
Instruction Decode (ID) holds its state fixed

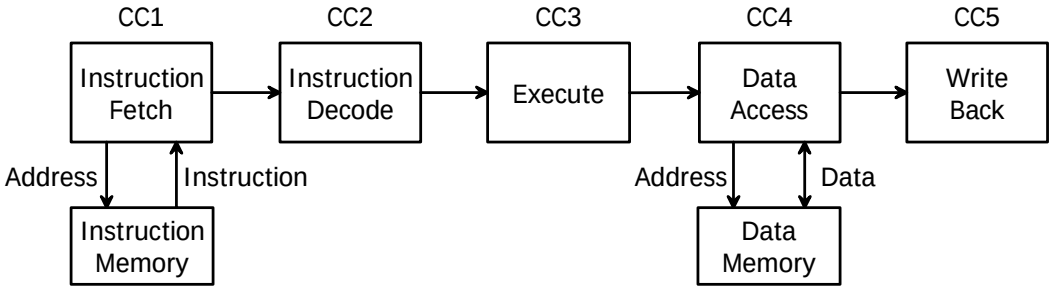
Clock Cycle4

I_1 moves to Memory Access (MEM)
Instruction Fetch (IF) holds its state fixed
Instruction Decode (ID) holds its state fixed
Execute (EX) holds its state fixed

Clock Cycle5

I_1 performs Write Back (WB) using instruction (IR) stored in IF stage
PC updated and stages IF, ID, EX, MEM are reset

There is no central system bus and instructions move from execution stage to execution stage. This description suggests the main possibility of improvement. DLXv2 is based on an industrial **assembly line**, which permits pipelining — in each stage new work begins when old work passes to the next stage.



Slides 4 – 7 DLX — Version 2

In DLXv2 each stage begins new work when the old work is passed to the next stage. The "work" is formalized as **Instruction State** — the 32-bit instruction and all intermediate results from earlier stages required to complete execution.

The operations on the first 5 instructions (in this ideal case) are:

CC1	I_1 enters Instruction Fetch (IF)
CC2	I_1 and its execution state move to Instruction Decode (ID) I_2 enters Instruction Fetch (IF)
CC3	I_1 and its execution state move to Execute (EX) I_2 and its execution state move to Instruction Decode (ID) I_3 enters Instruction Fetch (IF)
CC4	I_1 and its execution state move to Memory Access (MEM) I_2 and its execution state move to Execute (EX) I_3 and its execution state move to Instruction Decode (ID) I_4 enters Instruction Fetch (IF)
CC5	I_1 moves to Write Back (WB) I_2 and its execution state move to Memory Access (MEM) I_3 and its execution state move to Execute (EX) I_4 and its execution state move to Instruction Decode (ID) I_5 enters Instruction Fetch (IF)

The steps in execution can be visualized in the **Processor View** table. Five columns show the instruction content in one of the 5 executions stages, and the rows show the state in a given clock cycle.

clock cycle ↓ stage →

	IF	**ID**	**EX**	**MEM**	**WB**
1	I_1				
2	I_2	I_1			
3	I_3	I_2	I_1		
4	I_4	I_3	I_2	I_1	
5	I_5	I_4	I_3	I_2	I_1
6	I_6	I_5	I_4	I_3	I_2
7	I_7	I_6	I_5	I_4	I_3
8	I_8	I_7	I_6	I_5	I_4

After CC4, 5 instructions are being processed at one time and each instruction is in a different stage of execution.

A different presentation is the **Instruction View** showing the progress of each instruction in a given clock cycle.

instruction ↓ clock cycle →

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
I_1	IF	ID	EX	MEM	WB			
I_2		IF	ID	EX	MEM	WB		
I_3			IF	ID	EX	MEM	WB	
I_4				IF	ID	EX	MEM	WB
I_5					IF	ID	EX	MEM
I_6						IF	ID	EX
I_7							IF	ID
I_8								IF

The **performance advantage in DLXv2** is intuitively obvious — the CPU is working on multiple instructions at the same time and should complete the program in a shorter time. This is called the **ideal case** because it does not take account of **dependencies** between instructions that may cause the pipeline to **wait** for results of previous instructions before continuing with the next instruction.

The formal calculation of speedup follows from the observation that I_1 finishes after N=5 clock cycles, I_2 finishes after N=6 clock cycles, I_3 finishes after N=7 clock cycles and so on. In general, a program with **IC instructions** will complete in N=IC+4 clock cycles. The average CPI in the ideal case is then

$$CPI = \frac{\text{clock cycles}}{\text{finished instructions}} = \frac{IC+4}{IC} = 1 + \frac{4}{IC} \xrightarrow[IC>>4]{} 1$$

This means that on average, one instruction completes on every clock cycle.

Slides 8 – 10 **Pipelining — Functional Requirements**

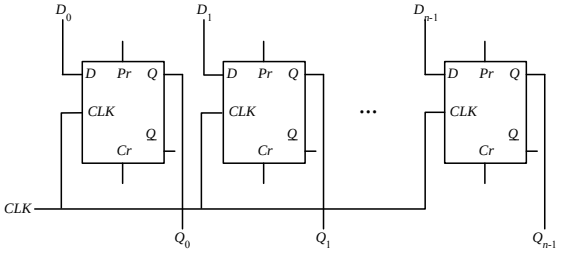
In DLXv2 each stage receives a new instruction on every clock cycle. The 5 stages cannot all hold the intermediate results for 5 instructions. So, each stage must transfer the **Instruction State** (the 32-bit instruction and all intermediate results) to the next stage after execution in that stage. For example,

IF stage
- Loads an instruction to **IR**
- Finds **NPC** for the next instruction
- Passes **IR** and **NPC** (intermediate results) to the **ID** stage

ID stage
- Stores the received **IR** and **NPC** for the incoming instruction
- Decodes **IR** to **A**, **B**, and **I**
- Passes **IR**, **NPC**, **A**, **B**, and **I** to the **EX** stage

In order to manage the temporary storage of these intermediate results, a **stage buffer** is placed between each pair of 5 pipeline stages.

Stage buffers are collections of **D-flip/flops** (also called **edge-triggered latches**) that perform **synchronous transfer**.

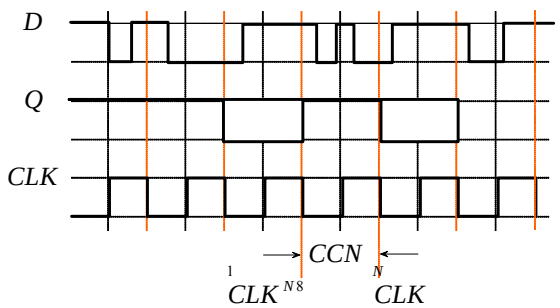


A buffer contains n D-flip/flops to store n bits:

Input D — the output of some digital system

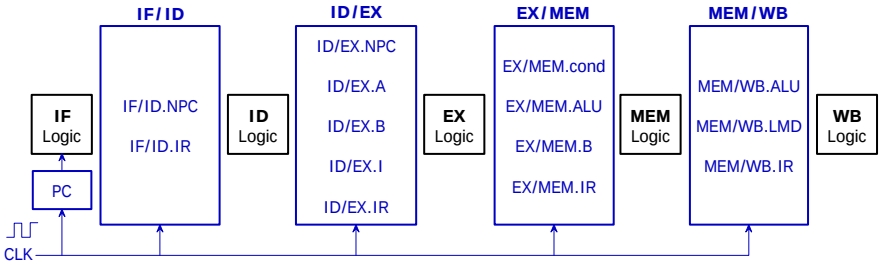
Output Q — samples D and changes only on the falling edge (1-to-0 transition) of the signal **CLK**

CLK (called a **Clock** or **Trigger**) is the same for all flip/flops — the n transitions are synchronized



Clock Cycle N defined as follows:
- CC N begins on CLK^{N-1}
  - Changes on Input D have no effect on the latch value or Output Q during the CC
- CC N ends on CLK^{N}
  - Latch samples Input D on falling edge of CLK
  - Stores instantaneous value of input D in latch
  - Forwards stored value to Output Q

DLXv2 redrawn with emphasis on the stage buffers appears as



A single system clock **CLK** is fed to all buffers, which synchronizes the transitions in the stages. In this version, the **PC** (address of next instruction) must also be transferred synchronously.

The logic elements in the 5 execution stages are built from two types of digital circuit

Combinational logic — output = function(present input)

Asynchronous memory — output = function(present input, past input)

The 4 stage buffers and the PC are built from another type of digital circuit

Synchronous sequential logic — output = function(present input, past input, external clock)

This arrangement insures **Synchronous Transfer**. When a clock cycle begins (falling edge of CLK) the synchronous stage buffers provide new inputs to the combinational logic and asynchronous memory in the pipeline stages. During the clock cycle the output of the stage logic changes asynchronously, depending on the propagation delays in each circuit. The period of the CLK signal (the period τ of the clock cycle) is chosen to insure that all of these asynchronous changes become stable before the end of the cycle. When the clock cycle ends (falling edge of CLK) the synchronous stage buffers sample their inputs (the outputs of the asynchronous circuits of the pipeline stage logic) store these values and transfer them to the next stage.

The stage buffers are named for the pipeline stages they buffer, using the dot-notation for members of a C structure:

IF/ID buffers NPC and IR from the IF stage for the ID stage.

ID/EX buffers the NPC and IR received from IF and the A, B, I results of stage ID.

EX/MEM buffers the IR and B results received from ID and the ALU and cond results of EX.

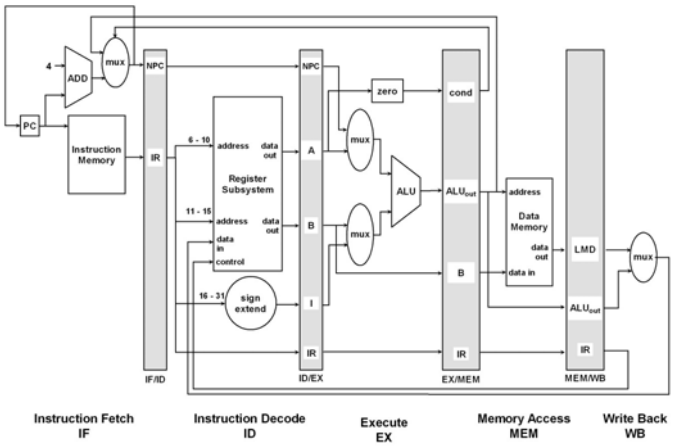
MEM/WB buffers the IR and ALU results received from EX and the LMD result of MEM.

Slides 11 – 13 DLX Drawing — version 2

The microarchitecture drawing of DLXv2 is shown on slide 11.

The stage buffers are shown in gray.

Notice that the multiplexor that chooses the next instruction address has been moved from the MEM stage to the IF stage. It is necessary to permit fetch of the next instruction on each clock cycle.



This version does not correctly handle branch instructions. That problem will be solved in DLXv6 (slides 45 – 48).

A formal specification of DLXv2 appears on slides 12 – 13.

Slide 14 Instruction Transfer Timing

As an example of synchronous transfer we consider the transfer of instructions across the pipeline. We write that a stage buffer "sees" an input while the asynchronous changes stabilize but the input has not been sampled or stored in the latch.

Trigger	Event	Transfer	Inputs
CLK 0	CC 1 begins	Memory ← PC(I_1)	IF/ID.IR "sees" Mem[PC(I_1)]
CLK 1	CC 2 begins	IF/ID.IR ← Mem[PC(I_1)] Memory ← PC(I_2)	ID/EX.IR "sees" Mem[PC(I_1)] IF/ID.IR "sees" Mem[PC(I_2)]
CLK 2	CC 3 begins	ID/EX.IR ← Mem[PC(I_1)] IF/ID.IR ← Mem[PC(I_2)] Memory ← PC(I_3)	EX/MEM.IR "sees" Mem[PC(I_1)] ID/EX.IR "sees" Mem[PC(I_2)] IF/ID.IR "sees" Mem[PC(I_3)]
CLK 3	CC 4 begins	EX/MEM.IR ← Mem[PC(I_1)] ...	MEM/WB.IR "sees" Mem[PC(I_1)] ...
CLK 4	CC 5 begins	MEM/WB.IR ← Mem[PC(I_1)]	Mem[PC(I_1)] controls Write Back

On CLK 0 clock cycle CC1 begins and a new value of PC for I_1 is presented to the IF logic. This value is used to read instruction I_1 from memory. During CC1 (between CLK 0 and CLK 1) the outputs of the memory circuits stabilize. Register IF/ID.IR "sees" these fluctuating signals during CC1 — when CLK 1 occurs these values have stabilized and are sampled and stored by the register latch. Following CLK 1 (in CC2) instruction I_1 is presented to the ID stage in the same way. At the end of CC2 (CLK 2) instruction I_1 is sampled and stored by ID/EX.IR.

In this way, instruction I_1 is stored in:

IF/ID.IR at the end of CC1
ID/EX.IR at the end of CC2
EX/MEM.IR at the end of CC3
MEM/WB.IR at the end of CC4

and can be used in the WB stage during CC5.

Slides 15 – 19 Simple 5-Instruction Program for DLX

Slide 16 shows the progress of simple 5-line program (slide 15). To avoid complications, this program contains no dependencies between lines (for that reason the program is trivial and accomplishes nothing).

Each column shows the actions performed in one pipeline stage in subsequent clock cycles. For example, column IF shows:

Fetch of I_1 from PC = 00 and preparation of NPC = 04 in CC1
Fetch of I_2 from PC = 04 and preparation of NPC = 08 in CC2
Fetch of I_3 from PC = 08 in and preparation of NPC = 0C in CC3
and so on.

Columns EX, MEM and WB show the specific actions taken for the different instruction types.

As an example of synchronous transfer (slide 17) we see that on:

CLK0 Memory ← PC =00 so that F/ID.IR "sees" Mem[00] and IF/ID.NPC "sees" 04 as inputs
CLK1 Memory ← PC =04 so that F/ID.IR "sees" Mem[04] and IF/ID.NPC "sees" 08 as inputs

Slide 18 shows the values "seen" by each stage buffer just before CLK4 ends CC4.
Slide 19 shows the values "seen" by each stage buffer just after CLK4 begins CC5.

Slides 21 – 22 Instruction Dependencies: Definitions

Up to now we have ignored dependencies between instructions. An **instruction dependency** occurs when the result of one instruction is needed to execute a later instruction. Naturally, dependencies are what give meaning to programs.

Hazard

A hazard exists when the processor runs smoothly but provides the wrong answers

Pipeline hazard

Several instructions are in various stages of execution in the CPU pipeline. If the pipeline uses a resource value before it is correctly updated by an earlier instruction, this is a pipeline hazard. This condition causes an error.

Example

PC ← NPC on each clock cycle
A branch instruction requires PC ← NPC + I
The correct evaluation of NPC + I is not available on the next clock cycle
The CPU fetches the wrong instruction

Hazard Types

Structural Hazard — caused by a conflict in the pipeline over access to a resource

Data Hazard — an instruction result is not ready when it is needed

Control Hazard — the branch address is not ready when it is needed

In order to prevent processing errors at run time, hazard conditions must be stopped. The simplest way to prevent the hazard is to pause the pipeline and wait for the required resource to become available. This is called a **WAIT STATE** or a **PIPELINE STALL**.

Stalls degrade processor performance because they add **stall clock cycles** to the instruction execution. This is formalized by considering the effect on CPI

$$CPI = \frac{\text{processing clock cycles (ideal) + stalled clock cycles}}{\text{completed instructions}}$$

$$= \frac{N^{ideal} + N^{stall}}{IC} = CPI^{ideal} + CPI^{stall} \xrightarrow[IC \to \text{ large on DLX}]{} 1 + CPI^{stall}$$

Similarly we characterize the performance degradation as

$$\text{performance degradation} = 1 - \frac{CPI^{ideal}}{CPI^{ideal} + CPI^{stall}} = \frac{CPI^{stall}}{1 + CPI^{stall}}$$

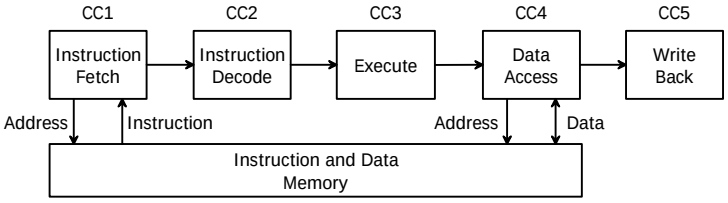
To avoid this degradation the next step is to eliminate the cause of the stall by analyzing the cause of the stalls and improving the implementation. This is the main professional activity of hardware architects.

Slides 23 – 25 Structural Hazards

A structural hazard is a conflict over access to resources. There are no structural hazards in DLXv2 but we must understand them to avoid introducing them while attempting to improve other aspects of the microarchitecture.

A typical structural hazard is the **unified cache hazard**. The DLX has a **split cache**: instructions are fetched from Instruction Memory (cache) in the IF stage and data is accessed in Data Memory (cache) in the MEM stage.

A pipeline with a unified cache has a single memory that is accessed from the IF stage and from the MEM stage:



Since instructions and data are in the same memory device, it is not possible to access data and fetch instruction on same clock cycle. This requires that instruction fetch waits (stalls) for 1 clock cycle on every data memory access (loads and stores). The implementation of the stall is seen in the processor view execution diagram as:

	IF	ID	EX	MEM	WB
CC1	I_1				
CC2	LW	I_1			
CC3	I_2	LW	I_1		
CC4	I_3	I_2	LW	I_1	
CC5	ϕ	I_3	I_2	LW	I_1
CC6	I_4	ϕ	I_3	I_2	LW
CC7		I_4	ϕ	I_3	I_2
CC8			I_4	ϕ	I_3
CC9				I_4	ϕ
					I_4

The Load Word (LW) instruction is fetched in CC2. The instructions proceed without a problem until CC5 when LW enters MEM and requires memory access. The data access of LW blocks the IF stage from accessing a new instruction. No instruction is fetched on CC5, symbolized by ϕ (null). There is no instruction to forward to ID on CC6, and instead IF forwards ϕ to represent **No Operation** (NOP). The NOP is also called a bubble. The NOP is forwarded to EX on CC7 and so on. No instruction completes execution on CC9.

To calculate the effect of this stall on CPI we expand the definition of CPI^{stall} as the **number of stall cycles per stall** multiplied by the **number of stalls per instruction**. There may be various **types of stall** so we must sum the stall cycles caused by different stall types i. There may also be various **instructions that cause stalls** so we expand stalls per instruction as **stalls of type i per instruction j** multiplied by the number of **instructions of type j per instruction** and sum over j.

$$CPI^{stall} = \frac{\text{stall cycles}}{\text{instructions}}$$

$$= \frac{\text{stall cycles}}{\text{stalls}} \times \frac{\text{stalls}}{\text{instructions}}$$

$$= \sum_{i=\text{type}} \left(\frac{\text{stall cycles}}{\text{stall}} \right)_i \times \left(\frac{\text{stalls}}{\text{instructions}} \right)_i$$

$$= \sum_{i,j} \left(\frac{\text{stall cycles}}{\text{stall}} \right)_i \times \frac{\text{stalls of type i}}{\text{instructions of type j}} \times \frac{\text{instructions of type j}}{\text{instructions}}$$

← This is the most general formula for CPI^{stall}.

$$= \sum_{i} \left(\frac{\text{stall cycles}}{\text{data stall}} \right)_i \times \left(\frac{\text{data stall}}{\text{instructions}} \right)_i \times \frac{IC_i}{IC}$$

← In the case of unified cache, there is only one stall type.

The factors in the expression for the unified cache stall are known:

There is 1 stall cycle for every stall of this type
There is 1 stall for every data access (load or store)
Load instructions are 25% of all instructions and store instructions are 15% of all instructions

Plugging in these values gives

$$CPI_{cache}^{stall} = \frac{1\text{ stall cycle}}{\text{stall}} \times \frac{1\text{ stall}}{\text{data memory load}} \times \frac{IC^{load}}{IC} + \frac{1\text{ cycle}}{\text{stall}} \times \frac{1\text{ stall}}{\text{data memory store}} \times \frac{IC^{store}}{IC}$$

$$= \frac{1\text{ stall cycle}}{\text{stall}} \times \frac{1\text{ stall}}{\text{data memory access}} \times \left(\frac{IC^{load}}{IC} + \frac{IC^{store}}{IC} \right)$$

$$= \frac{1\text{ stall cycle}}{\text{stall}} \times \frac{1\text{ stall}}{\text{data memory access}} \times \left(\frac{0.25\text{ loads}}{\text{instruction}} + \frac{0.15\text{ stores}}{\text{instruction}} \right)$$

$$= 0.40 \frac{\text{stall cycles}}{\text{instruction}} \Rightarrow CPI = CPI^{ideal} + CPI^{stall} = 1.40$$

Most modern CPUs use split cache to avoid this stall. The performance degradation associated with $CPI^{stall} = 0.4$ is

$$\text{degradation} = \frac{CPI^{stall}}{1 + CPI^{stall}} = \frac{0.4}{1.4} = 0.29 = 29\%$$

Slide 26 Data Hazards

A data hazard occurs when an instruction result is not ready when it is needed. If the execution continues without correction then operations are performed in the wrong order. The types of data hazard are classified and named for the correct order of operations.

Read After Write (RAW)

Correct **I_2 reads** register **after I_1** writes to it
Hazard I_2 reads register before I_1 writes to it
I_2 uses incorrect value

Write After Write (WAW)

Correct **I_2 writes** to register **after I_1** writes to it
Hazard I_2 writes to register before I_1 writes to it
Incorrect value stays in register

Write After Read (WAR)

Correct **I_2 writes** to register **after I_1 reads** it
Hazard I_2 writes to register before reads I_1 it
I_1 uses incorrect value

Read After Read (RAR)

No hazard — reads do not affect registers

RAW hazards

A DLX register is updated in stage WB. The next instruction may attempt to read the same register in stage ID before it is updated. This is a possible hazard that must be avoided.

WAW hazards cannot occur

The DLX writes to memory and registers in the order of the instructions. Memory is always updated in MEM and registers are always updated in WB. Since all updates are performed in the order of execution, I_2 cannot perform either WB or MEM before I_1 performs WB or MEM. There is no possible error in the order of writing.

WAR hazards cannot occur

Loads are performed in MEM and registers are read in ID. A store is performed in MEM and registers updated in WB. Therefore I_2 cannot perform a register update in WB or a memory write in MEM before I_1 performs its register read in ID or a memory read in MEM.

Although WAW and WAR hazards are not possible in DLX, we must be sure not to create them while trying to improve the pipeline.

As an example of RAW hazards we consider the program

```
I1  ADD R1,R2,R3        I1 has R1 as destination
I2  SUB R4,R5,R1
I3  AND R6,R7,R1        I2 – I4 have R1 as source
I4  OR  R8,R9,R1
```

If the program runs without correction

	IF	ID	EX	MEM	WB
CC1	ADD				
CC2	SUB	ADD			
CC3	AND	SUB	ADD		
CC4	OR	AND	SUB	ADD	
CC5		OR	AND	SUB	ADD
CC6			OR	AND	SUB
CC7				OR	AND
CC8					OR

I_1 updates **R1** in **WB** during **CC5**

I_2 reads old **R1** in **ID** during **CC3**

I_3 reads old **R1** in **ID** during **CC4**

I_4 reads **R1** in **ID** during **CC5**

I_2 and I_3 read the incorrect value of R1 in CC3 and CC4. I_4 reads the correct updated value of R1 in CC5. This is because the update of R1 occurs at the beginning of CC5 and the final read (sample and latch) of R1 for I_4 in ID occurs at the end of CC5. See slide 29 for a detailed analysis.

To prevent the register-register RAW hazard, **DLXv2 must stall the pipeline** for 2 clock cycles. This is seen in the processor view execution table

	IF	ID	EX	MEM	WB
CC1	ADD				
CC2	SUB	ADD			
CC3	AND	SUB	ADD		
CC4	AND	SUB	ϕ	ADD	
CC5	AND	SUB	ϕ	ϕ	ADD
CC6	OR	AND	SUB	ϕ	ϕ
CC7		OR	AND	SUB	ϕ
CC8			OR	AND	SUB
				OR	AND
					OR

The DLX control system must be able to identify all hazards and insert stall cycles when necessary.

Wait states during CC3 and CC4

ID/EX freezes internal state on **SUB**
IF/ID freezes internal state on **AND**
ID performs **NOP** (no operation) to avoid reading old value of **R1**
ID/EX passes ϕ (**NOP**) to **EX**

Continuation — no hazard in CC5

WB performed at start of clock cycle
SUB result locked at end of clock cycle

The pipeline stall in instruction view is

Clock Cycle ⟶

	1	2	3	4	5	6	7	8
ADD R1,R2,R3	IF	ID	EX	MEM	WB			
SUB R4,R5,R1		IF	ID	ID	ID	EX	MEM	WB
AND R6,R7,R1			IF	IF	IF	ID	EX	MEM
OR R8,R9,R1						IF	ID	EX

SUB is held in ID for CC3 and CC4 (ID/EX is frozen on SUB).

The performance degradation is found from

$$CPI^{stall} = \frac{\text{stall cycles}}{\text{stalls}} \times \frac{\text{stalls}}{\text{instruction type}} \times \frac{\text{instruction types}}{\text{instruction}}$$

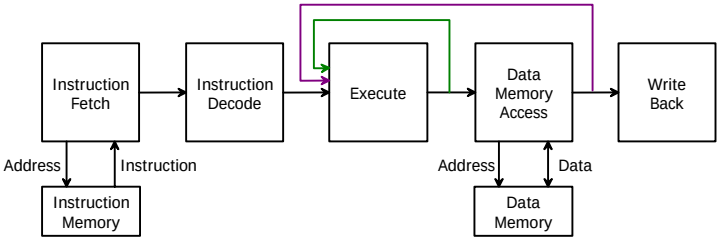
$$= \frac{\text{2 stall cycle}}{\text{stall}} \times \frac{\text{0.5 register dependencies}}{\text{ALU instruction}} \times \frac{\text{0.4 ALU}}{\text{instruction}}$$

$$= 2 \times 0.5 \times 0.4 \frac{\text{cycles}}{\text{instruction}} \quad \Rightarrow \quad CPI = 1.4 \Rightarrow \text{(29\% degradation)}$$

and is seen to be serious.

Slides 32 – 34 Forwarding or Bypass (DLX Version 3)

The register-register RAW stall is prevented in **DLX Version 3** by a technique called **forwarding** (or bypass). The basic idea is that the **ADD** instruction calculates its ALU result in CC3 but does not update R1 until CC5. If we force an early write to R1 we may create new data hazards. Instead, we **allow** the **SUB** and **AND** instructions to read **incorrect values** in ID (normal register reads) but we **replace the values** in EX when the instructions need them. We must provide the correct value from **EX/MEM.ALU** and **MEM/WB.ALU** directly to the ALU input. DLXv3 has new data paths from the buffers that store the result of instruction 1 directly to the ALU.

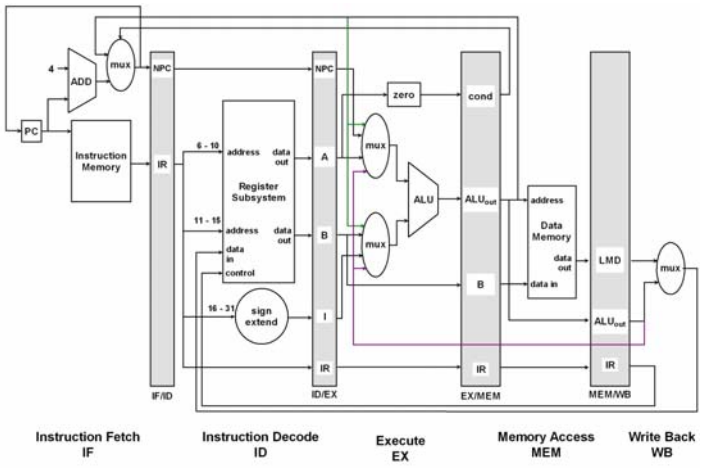


The processor view execution diagram is

	IF	ID	EX	MEM	WB
CC1	ADD				
CC2	SUB	ADD			
CC3	AND	SUB	ADD		
CC4	OR	AND	SUB	ADD	
CC5		OR	AND	SUB	ADD
CC6			OR	AND	SUB
CC7				OR	AND
CC8					OR

The arrows show the forwarding of the result of ADD to the ALU.

The drawing of the DLXv3 microarchitecture (slide 33) shows the new data paths added.



The MUXes in EX have two new choices: EX/MEM.ALU_{out} (the previous ALU result) and MEM/WB.ALU_{out} (the ALU result from before the previous). Either can replace A or B in forwarding.

The instruction view execution diagram is

Clock Cycle ⟶

	1	2	3	4	5	6
ADD R1,R2,R3	IF	ID	EX	MEM	WB	
SUB R4,R5,R1		IF	ID	EX	MEM	WB
AND R6,R7,R1			IF	ID	EX	MEM
OR R8,R9,R1				IF	ID	EX

The processor moves the state of the **ADD** instruction from buffer to buffer

SUB needs the **ALU result** in **CC4**

ADD provides the **ALU result** from **EX/MEM.ALU**

AND needs the **ALU result** in **CC5**

ADD provides the **ALU result** from **MEM/WB.ALU**In DLXv3 there are no stalls for register-register dependencies and so $CPI^{stall} = 0$.

Slides 35 – 38 Register-Load RAW Dependencies in DLXv3

Another type of RAW dependency occurs when a load to a register does not complete before the register is needed. This case differs from the register-register load dependency because the required result is not available in the clock cycle following the EX stage.

Slide 35 replaces the instruction I_1 ADD R1, R2, R3 with the instruction I_1 LW R1, 32(R2). In the case of ADD, the value that is written to R1 in CC5 is available after CC3 and can be provided directly to the ALU for I_2 in CC4 and I_3 in CC5. In the case of LW, the address of the load is available after CC3, but the loaded data is not available until after CC4.

Slide 36 shows the new forwarding data paths in **DLX version 4** and the timing of the forwarding in processor view. Since the loaded data in only available for forwarding in CC5 the SUB instruction following the load must **stall for one cycle** in the ID stage. Slide 37 shows the new data path on the DLXv4 microarchitecture — the red line from LMD (load memory data) back to the multiplexors in EX.

Slide 38 shows the **forwarding** and the **stall** in instruction view. The data loaded by LW in MEM in CC4 is made available to the SUB in CC5. The SUB must wait in the ID stage during CC4 causing the stall.

The CPI^{stall} calculation requires:

1 stall cycle per stall

0.25 load instructions per program instruction (IC_{load} / IC)

The number of stalls per load instruction is just the number of times a loaded data operand is an ALU source in the next instruction. This happens about 50% of the time — in the other 50% another operand is usually loaded to a different register.

CPI^{stall} = 1 stall cycle per stall × 0.5 stalls per load × 0.25 loads per instruction = 0.125.

We will see (slide 41) that this performance degradation can be improved with a compiler optimization.

Slides 39 – 40 Register-Store RAW Dependencies in DLXv4

Another type of RAW dependency occurs when an ALU result is written to memory by the next instruction (slide 39). This dependency can also be treated with forwarding as seen in slide 40.

In **DLX version 5** a new multiplexor is added as the data input for memory store instructions. Normally the data written to memory is put into B during the ID stage of the SW instruction. When the program performs an ALU operation and stores the result immediately, the new multiplexor in the MEM stage can choose MEM/WB.ALU_{out} (the value that updates the register in the WB stage) instead of B (the previous value).

Slide 41 Compiler Scheduling to Prevent RAW Hazards

On the first pass a compiler that receives the C instructions

```
I = I + 123;
J = J - 567;
```

Will first compile

```
I = I + 123; →    LW   R2, I
                  ADD R2,R2, #123
                  SW   I, R2
```

and then compile

```
J = J - 567; →    LW   R3, J
                  SUB R3, R3, #567
                  SW   J, R3
```

If the program runs this way, there will be a stall cycle following each LW instruction.

An optimizing compiler will make a second pass and reschedule (rearrange) the instructions in the following order:

```
I1 LW   R2, I
I2 LW   R3, J
I3 ADD R2,R2, #123
I4 SW   I, R2
I5 SUB R3, R3, #567
I6 SW   J, R3
```

The new program order does not change the numerical results because the new order respects the data dependencies in the original version. But as can be seen in the execution diagrams on slide 41 this version has no stall cycles. Instruction I_2 is fetched before I_3 so that when I_3 reaches EX the value of R2 loaded by I_1 is available for forwarding. When I_5 needs the value of R3 it is already in the register.

The first version runs in 12 CC and the second version runs in 10 CC.

Slides 42 – 44 DLX Control Hazard

In versions 2 – 5 of the DLX, PC ← NPC on each clock cycle so that there is a new PC for a new instruction fetch in every clock cycle. The **control hazard** occurs whenever there is a conditional branch instruction because the next instruction (taken or not taken) is not yet known.

Definitions

Fall-through — the instruction immediately following a branch in the program listing

Target — the instruction pointed to by the branch

Branch taken — the condition for the branch is true and the branch causes a jump to the target instruction

Branch non-taken — the condition for the branch is false and the branch continues to the fall-through instruction

The stages of execution for a branch are

CLK	CC	Latched State	Action during CC
0	1	`Memory ← PC(branch)`	IF/ID.IR "sees" IR(branch) and PC(branch)
1	2	`IF/ID.IR ← branch`	Decode of branch instruction, NPC, I
2	3	`ID/EX.NPC ← NPC` `ID/EX.I ← I`	Calculate address NPC + I and cond
3	4	`EX/MEM.ALU ← ALU` `EX/MEM.cond ← cond`	PC "sees" correct address via MUX using cond to choose NPC or NPC+I
4	5	`PC ← branch address`	IF/ID.IR "sees" correct next instruction

When the instruction fetched on CC1 is a branch, the correct address for the next instruction (target or fall-through) is not known until the end of CC4 and cannot be fetched until CC5.

The simplest treatment for the DLX control hazard is to stall and **flush the pipeline**:

		1	2	3	4	5	6	7	8	9
I_1	BEQZ R1,I_T	IF	ID	EX	MEM	WB				
I_2	Fall-Through		IF	ϕ	ϕ	IF	ID	EX	MEM	WB
I_3				ϕ	ϕ					
...	...	...	...	...	...	...	...	...	...	...
I_T	Target					IF	ID	EX	MEM	WB

CC1 fetch BEQZ

CC2 BEQZ is identified in ID as a conditional branch
The fall-through is fetched in IF (automatically)

CC3 The fall-through is flushed (deleted) from the pipeline
EX calculates the target address in the ALU and evaluates the condition

CC4 PC "sees" correct next address from EX/MEM.ALU and EX/MEM.cond

CC5 PC latches correct next address and fetches fall-through or target (depends on cond)

The performance degradation is severe:

CPI^{stall} = 3 CC per stall × 1 stall per branch × 0.20 branches per instruction = 0.60

This is equivalent to a 38% degradation in performance.

The branch stall will be treated in 3 steps: versions 6a, 6b and 6c.

Slides 45 – 48 DLX Version 6

The first enhancement (**DLXv6a**) is to add a special CLK to the PC to prevent waiting during CC4. At the end of CC3 an extra CLK pulse triggers PC to read from PC+4 or EX/MEM.ALU (depending on the value in EX/MEM.cond). The new instruction is now fetched in CC4 (instead of CC5). This reduces the number of stall cycles from 3 to 2 as shown on slide 45.

The second enhancement (**DLXv6b**) is more complex. Instead of computing the target address in the general ALU (stage EX), a special dedicated ADDER is placed in the ID stage. Also the comparator to set the cond flag is moved to ID. Instead of storing NPC in ID/EX for use in the general ALU, the special ALU adds IF/ID.NPC to the value that will be stored in I. Buffer ID/EX has two new registers, NNPC = NPC + I and cond. These values are now available to the IF stage at the end of the ID stage of the conditional branch. This reduces the number of stall cycles from 2 to 1 as shown on slide 46. In the execution diagram we see that if the condition is false then the fall-through instruction is cancelled in CC2 and then fetched again in CC3.

The third enhancement (**DLXv6c**) is implemented in the controller. If the condition is false then the fall-through instruction is the correct instruction, so it is not cancelled in CC2. The pipeline can continue in this case as seen on slide 47. This means that there is a stall of 1 clock cycle when the branch is taken and no stall when the branch is not taken. The complete **microarchitecture** drawing for DLXv6c is shown on slide 48.

Slides 49 – 52 Version 6c Branch Processing

The execution of a **taken conditional branch** begins on CC1 with the fetch of the BEQZ instruction. Slide 49 shows PC providing the address of the BEQZ instruction and to the adder that produces the address of the fall-through instruction. The PC (a buffer) and IF/ID.NPC "see" the fall-through address as their inputs (but do not latch the new address until triggered by CLK). ID/IF.IR "sees" the instruction as its input. On the CLK trigger, PC, IF/ID.NPC and IF/ID.IR sample and store their inputs.

On CC2 (slide 50) the adder in ID calculates the target address from IF/ID.NPC and the immediate (that will be stored in I at the end of the clock cycle). The address of the fall-though is provided to memory which provides the fall-through instruction to the input of IF/ID.IR (fetch of fall-through). At the end of the clock cycle (CLK trigger) the address of the target and cond are latched into ID/EX.NNPC and ID/EX.cond

On CC3 (slide 51) the fall-through instruction is flushed from ID, which now holds a NOP. PC "sees" the address of the target as input. The system controller quickly recognizes the conditional branch (slide 52) and sends a second "special" CLK trigger to the PC (since the pipeline was flushed this does not have any effect on the rest of the CPU). Without wasting any more time, the PC latches the address of the target instruction — this causes the fetch of the target instruction in CC3.

On CC4 the target instruction is latched into IF/ID.IR and the execution continues.

Slide 53 Branch Performance of Version 6c

The treatment of conditional branch in DLXv6c is called **Predict-Not-Taken** because it provides better performance on a branch not taken (no pipeline stall):

Branch taken — Flush entire pipeline
Branch not taken — Continue instruction in IF

From statistics on programs from SPEC CINT the actual probabilities are:

Not taken 33%
Taken 67%

Calculating CPI^{stall} for branches

$$CPI^{stall} = \frac{\text{stall cycles}}{\text{stalls}} \times \frac{\text{stalls}}{\text{instruction type}} \times \frac{\text{instruction types}}{\text{instruction}}$$

$$= \frac{\text{stall cycles}}{\text{taken branch}} \times \frac{\text{taken branch}}{\text{branch instruction}} \times \frac{IC^{branch}}{IC}$$

$$= 1 \times 0.67 \times 0.20 \frac{\text{cycles}}{\text{instruction}} = 0.13 \frac{\text{cycles}}{\text{instruction}}$$

$$CPI = 1.13 \Rightarrow \text{(12\% degradation)}$$

This value is an improvement over the initial value of 38% degradation for pipeline flush, but is still in need of improvement. Large CPUs use a complex technique of branch prediction (slide 61) not implemented for DLX. About half of all transistors (and chip "real estate") in an Intel x86 CPU are used for branch prediction.

Slides 54 – 56 DLXv6c Pipeline

The DLXv6c pipeline is shown on slide 54. The drawing includes the data paths added for forwarding and a Floating Point Unit (FPU).

Execution of float instructions is similar to integer execution, except that instructions are sent to the **FPU** instead of the EX stage that contains an **integer ALU**. The FPU is complex because floating point is a more complex data structure than 2's complement signed integers.

The **forwarding mechanisms** treat ALU-to-ALU RAW data dependencies with no stall and Load-to-ALU RAW data dependencies with a 1 CC stall. The ALU-to-Store data dependency is handled with no stall if the Store follows the ALU immediately but causes a stall if another instruction comes between ALU and Store. All other data dependencies cause a stall until the required data appears in its register following a write-back. These situations are summarized on slides 57 – 60.

A **formal specification** of DLXv6c is given on slides 55 – 56.

Slides 57 – 60 Forwarding

Slide 57 shows the instruction view execution table for **ALU-to-ALU** forwarding. The arrows show the forwarding data paths used to provide ALU result back to the ALU, instead of the previous values read during the ID stage. The third instruction requires forwarding from both the first and second instruction. The second instruction forwards its result directly from EX after

the ALU operation completes. The first instruction forwards its result from MEM where it stores it to be written to R1 in WB.

Slide 58 shows the instruction view execution table for **Load-to-ALU** forwarding.

In the first case the loaded data is used in the next two instructions, causing a 1 CC stall between LW and the second instruction. The third instruction requires no forwarding — it reads R1 correctly in ID following the writeback.

In the second case we see another 1 CC stall between LW and ADD. But notice that the second and third instructions can be exchanged with no change to result. The rescheduled program is seen to execute without a stall.

Slide 59 shows the instruction view execution table for **ALU-to-Store** forwarding. There is no stall when the Store instruction follows the ALU instruction immediately. But as seen in the following diagram, inserting an unrelated instruction between the ALU and the Store creates a 1 CC stall. There is no forwarding mechanism to bring the ALU result from MEM to the Store input. The Store must stall until R1 is written to the register in WB.

Slide 60 shows the instruction view execution table for **ALU-to-Branch** operations. There is no forwarding mechanism to bring the ALU result from EX to the NNPC input in ID. The branch must stall for 2 CC until R1 is written to the register in WB. This case is best handled by compiler optimization — placing independent work between the ALU instruction and the branch.

Slide 61 Improvement by Re-Scheduling in DLXv6c

The result of compiler optimization is shown on slide 61 for the sample program given on slide 8 of chapter 5. The first version of the program loads the first two operands and performs an ALU operation, then another load and another ALU, and so on. At the end of the Load/ALU block it updates the loop index R1 and performs a branch. Each LW-ALU pair causes a 1 CC stall. The ALU-Branch causes a 2 CC stall.

The optimized code issued by the compiler after a second pass first updates the loop index R1, then performs all of the loads at once, then all the ALU instructions at once, and then the Branch. All of the data dependencies have been "distanced" from one another so that the latency (number of CC between the start of an operation and the availability of the result) is "hidden" by other work being performed. This version runs with no stall cycles (except for the 1 CC stall between BNEZ and the SUBI in the next loop iteration because of predict-not-taken), saving 5 CC per loop.

The speedup from this optimization is significant. The loop runs 256 times and so we may neglect the first instruction (ADDI R1, R0, #400), which runs only once. In each loop iteration, 10 instructions (SUBI to BNEZ) run.

CPI_1^{stall} = 256 x (3 CC for LW-ALU + 2 CC for ALU-Branch + 1 CC for Branch) / (256 x 10) = 0.60

CPI_2^{stall} = 256 x (1 CC for Branch) / (256 x 10) = 0.10

$$S = \frac{CPI \times IC \times \tau}{CPI' \times IC' \times \tau'} = \frac{CPI}{CPI'} = \frac{1 + CPI^{stall}}{1 + CPI^{stall}{}'} = \frac{1.60}{1.10} = 1.45$$

which is a 45% improvement.

Slides 62 – 64 Improvement by Parallel Threads in DLXv6c

Slide 62 shows a sequential (non-threaded) implementation of the source code

```
for ( i = 0 ; i < 100; i++ ){
      c[i] = a[i] + b[i];
      d[i] = a[i] - b[i];
}
```

The performance is found as follows:

- The loop runs 400 / 4 = 100 iterations
- Each iteration includes 9 instructions
- Each iteration (except the last) includes 4 stall cycles
  LW $\rightarrow$ ADD = 1, SRT $\rightarrow$ BNEZ = 2, BNEZ $\rightarrow$ L1 = 1
- The last iteration includes 3 stall cycles (no stall on BNEZ $\rightarrow$ J)
- CPI_{stall} = 4 CC / 9 instructions = 4/9 CC per instruction $\Rightarrow$ CPI = 1 + 4/9 = 13/9
- Loop CC = 100 iterations $\times$ 9 instructions per iteration $\times$ 13/9 CC per instruction = 1300 CC
- Before we count the first instruction we count 4 CC = CC1 to CC4
- Outside the loop there are 3 instructions with no stalls = 3 CC
- Total CC = 4 + 2 (ADDIs) + 1300 – 1 (no stall on last BNEZ) + 1 (J) = 1306

Slide 63 shows a division into two threads by **DATA DECOMPOSITION**:

- Thread 0 handles iterations 0 to 50 and Thread 1 handles iterations 50 to 100.
- The code for each thread is the same as the sequential code (except for the start and stop values).
- Loop CC = 50 iterations $\times$ 9 instructions per iteration $\times$ 13/9 CC per instruction = 650 CC
- Total CC = 656 CC
- S = 1306 CC / 656 CC = 1.99
- The 6 overhead CC run on both threads and so are sequential.

Slide 64 shows a division into two threads by **FUNCTIONAL DECOMPOSITION**:

- Thread 0 handles the ADD and Thread 1 handles the SUB. Both threads run 100 times
- The stalls are the same but now there are 7 instructions per loop iteration
- CPI = 1 + 4/7 = 11/7
- Total CC for loop = 100 iterations $\times$ 7 instructions $\times$ 11/7 CC = 1100
- Total CC = 1106 CC
- S = 1306 CC / 1106 CC = 1.18
- Most of the instructions are run on both threads and so functional decomposition produces a much smaller speed-up.

Data decomposition is usually more efficient that functional decomposition, but is often more difficult to implement. Consider for example the data decomposition of the following code:

```
for ( i = 0 ; i < 100; i++ ){
      sum = sum + a[i];
}
```

This can be split into threads as

Thread 0

```
sum_12 = 0

for ( i = 0 ; i < 50; i++ ){
      sum_0 = sum_0 + a[i];
}

while (sum_12 == 0){
// spin loop: wait for value
}

sum = sum_0 + sum_12
```

Thread 1

```
for ( i = 50 ; i < 100; i++ ){
      sum_1 = sum_1 + a[i];
}

sum_12 = sum_1 ;
```

Most problems in parallel programming are more complex than these.

Slides 65 – 68 General Branch Prediction

The branch statistics from SPEC CINT are

Branch not taken	33%
Branch taken	67%

So that most branch instructions are taken. This makes sense because branches are used to build loops and usually run more than once.

Branch prediction is an advanced technique that improves the execution of taken branches at the cost of doubling the complexity of the CPU. It is not implemented in the DLX model but is used in most modern RISC processors and in Intel x86 CPUs since the original Pentium (1995).

The extra hardware includes a **BRANCH PREDICTOR** that records statistics on branch instructions:

Source address of branch instruction
Target address
Taken/not-taken

Using these statistics the hardware can predict branch behavior based on previous behavior.

If branch prediction were added to the DLX it would appear as on slide 63:

The IF stage would include the **branch predictor** that assembles a **database** of branch instructions indexed by the source address (unique location of branch instruction in memory at run time). The entry for each record is information on its history of taken/no-taken and the target address it points to.

When a recognized branch instruction is fetched (on the 2nd or 3rd time the branch runs) the branch predictor makes a **prediction** of the next instruction address to fetch (target if taken or fall-through if not-taken). The IF fetches the predicted instruction.

When the branch instruction enters ID it is executed in the usual way to **validate** the prediction. If the prediction was correct then execution continues and the statistics are updated in the branch prediction database. If the prediction was incorrect — **misprediction** — then the pipeline is flushed, the correct instruction is fetched and the statistics are updated in the branch prediction database.

Slide 67 shows the execution diagrams for the first and second execution of a certain branch for a simple loop (highly simplified model). The branch is unknown on the first fetch and is predicted not-taken (no other guess can be made). The branch is taken and the misprediction is handled. On the second execution the branch is recognized and predicted as taken.

Slide 68 shows a very simple program with BNEZ as the control instruction for a loop. On the first execution of BNEZ it is unknown and mispredicted as not-taken. On the next N – 2 loop iterations the BNEZ is correctly predicted as taken. On the last loop iteration BNEZ is mispredicted as taken — the loop has finished and the branch is not-taken. Out of N iterations of B instructions per iteration, there are only 2 CC of stall, which is negligible for large N.

Slides 68 – 70 More Compiler Optimizations

Another compiler optimization is **common sub-expression elimination**. When the compiler encounters instructions of the type

```
B = 10*(A/3);
C = (A/3)/4;
```

It calculates `(A/3)` into a register and then uses the register value in later calculations.

First Pass Compilation

```
LW R1,A
ADDI R2,R0,#3
DIV R1,R1,R2
ADDI R2,R0,#10
MULT R1,R1,R2
SW B,R1
LW R1,A
ADDI R2,R0,#3
DIV R1,R1,R2
ADDI R2,R0,#4
DIV R1,R1,R2
SW C,R1
```


Second Pass Compilation

```
LW R1,A
ADDI R2,R0,#3
DIV R1,R1,R2
ADDI R2,R0,#10
MULT R3,R1,R2
SW B,R3
ADDI R2,R0,#4
DIV R3,R1,R2
SW C,R3
```

Loop unrolling is an optimization in which the compiler replicates instructions instead of implementing a loop. This is considered "inelegant" in high-level programming, but may be more efficient to implement because it eliminates the overhead of testing loop control variable.

Similarly, **inlining** is an optimization that replaces a procedure call with the code of the procedure or macro.

First Pass Compilation

```
00  ADDI R2,R0,#0x05
04  ADDI R1,R0,#0x08
08  LW R3,0x1000(R1)
0C  JAL 10
10  SW 2000(R1),R3
14  SUBI R1,R1,#0x04
18  BNEZ R1,-0x14
1C  ADDI R2,R0,#3
20  ADD R3,R3,R2
24  JR R31
```


Second Pass Compilation

```
00  ADDI R2,R0,#0x05
04  LW R3,0x1008(R0)
08  ADD R3,R3,R2
0C  SW 2008(R1),R3
10  LW R3,0x1004(R0)
14  ADD R3,R3,R2
18  SW 2004(R1),R3
1C  ADDI R2,R0,#3
```

In the first pass an operand is loaded to R3 and passed to the procedure at address 20. The ALU result is returned and stored in the instruction at address 10. The index is updated and the program loops back to LW at address 08. The loop runs twice. The Load-ALU pairs and ALU-Branch pairs cause many stall clock cycles for a short program.

Slide 71 More Hardware Optimizations

After RISC became established in the 1990s, so-called second generation RISC introduced new and complex hardware systems that further enhance pipeline efficiency and lower CPI below 1.

Superscaling

A superscalar CPU operates 2 or more pipelines in parallel. Instructions without dependencies can execute in parallel so that more than one instruction completes on each CC and CPI < 1. This method is used in most RISC processors and in Intel x86 processors since the original Pentium (1995).

Dynamic Scheduling

Instead of relying on compiler optimization for efficient instruction scheduling, the processor performs dynamic instruction scheduling at run time. The CPU chooses instructions with no unresolved dependencies from a pool of instructions, a very efficient method when combined with superscaling. This technique has been used in IBM mainframes since 1967 and is used in all Intel x86 processors since the Pentium II (1997).

Register Aliasing

Independent tasks and threads require access to the same logical registers (R0, R1, … as defined in ISA). Many processors provide physical registers than can be allocated per task from a large register pool. Then, multiple tasks can use the same logical register in parallel without causing data dependencies. This technique has been used in IBM mainframes since 1967 and is used in all Intel x86 processors since the Pentium II (1997).

Instruction Predication

A predicated instruction can be run or just cancelled according to a predicate flag associated with the instruction. Predication flags can be set from standard test-and-set instructions (SLT, SGT, SEQ, …). An IF-THEN-ELSE structure can be implemented as

```
if (test) set flag
(flag) action for true        // cancelled if flag = false
(!flag) action for false      // cancelled if flag = true
```

This eliminates the use of conditional branch instructions to jump over instructions and saves many stall clock cycles. Predication is used on the ARM RISC CPU, some microcontrollers and the Intel Itanium.