**PIPELINING**

Bizim uygulayacağımız pipeliningte her cycleda 1 stage bitecek. Yani her cycleda 1 instruction bitiyor olacak.

Pipelining provides a method for executing multiple instructions at the same time.

A picture containing diagram

Description automatically generatedGraphical user interface, application

Description automatically generated

Pipelining doesn’t help latency of single task, it helps throughput of entire workload

Pipeline rate limited by slowest pipeline stage

Multiple tasks operating simultaneously using different resources

Potential speedup = Number pipe stages

Unbalanced lengths of pipe stages reduces speedup

Time to “fill” pipeline and time to “drain” it reduces speedup

Stall for Dependences

Table

Description automatically generated

Table

Description automatically generatedSingle Cycle vs. Pipelined Execution

Pipelining Speedup

If the stages are perfectly balanced (require same time):

Time between instructionspipelined (Time for 1 stage) =

Potential speedup = Number of pipeline stages

In previous example, 3 instructions takes 14 ns.

If we would add 1000 instructions then each instruction will add 2 ns to the total execution time:

Total execution time pipelined = 14 + 2000 = 2014 ns

Total execution time nonpipelined = 1003 \* 8 = 8024 ns

8024 / 2014 = 3.98 ~ 8 / 2

The Five Stages of the Load Instruction

Diagram

Description automatically generated with medium confidence

Ifetch : Instruction fetch

* Fetch the instruction from the instruction memory

Reg/Dec : Registers fetch and instruction decode

Exec : Calculate the memory address

Mem : Read the data from the data memory

Wr : Write the data back to the register file

Pipelined Execution

Graphical user interface, application

Description automatically generated

We should assume each stage takes same time. Even dcd stage may require less time.

Single Cycle, Multiple Cycle, vs. Pipeline

Table

Description automatically generated

1 instruction can run faster in single cycle because we use more stage in Reg (for example) to be synchronized.

Cycle of the pipelined processer is much faster than the cycle of the single cycle processor.

Graphically Representing Pipelines

Diagram, schematic

Description automatically generated

Why Pipeline?

Suppose:

* 100 instructions are executed
* The single cycle machine has a cycle time of 45 ns
* The multicycle and pipeline machines have cycle times of 10 ns
* The multicycle machine has a CPI of 3.6

Single Cycle Machine:

* 45 ns/cycle x 1 CPI x 100 inst = 4500 ns

Multicycle Machine:

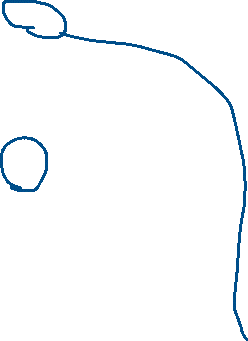
* 10 ns/cycle x 3.6 CPI x 100 inst = 3600 ns

Ideal pipelined machine:

* 10 ns/cycle x (1 CPI x 100 inst + 4 cycle drain) = 1040 ns
* After the 4th cycle, each instruction finishes execution.

Diagram

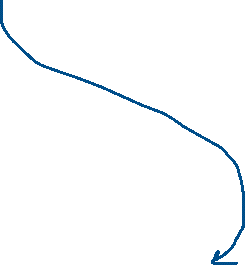
Description automatically generated



Ideal pipelined vs. single cycle speedup:

* 4500 ns / 1040 ns = 4.33

What has not yet been considered?



Can pipelining get us into trouble?

* Yes: Pipeline Hazards
  + Structural Hazards: Attempt to use the same resource (hardware unit) two different ways at the same time
    - E.g., two instructions try to read the same memory at the same time
    - Using different memory
      * Separate memory as data memory and instruction memory if two instructions’ stages are both try to do sth with memory
  + Data Hazards: Attemp to use item before it is ready
    - Instruction depends on result of prior instruction still in the pipeline
      * add r1, r2, r3
      * sub r4, r2, r1
  + Control Hazards: Attempt to make a decision before condition is evaluated
    - Branch instructions
      * beq r1, r2, loop
      * add r3, r4, r5

Can always resolve hazards by waiting

* pipeline control must detect the hazard
* take action (or delay action) to resolve hazards

Single Memory is a Structural Hazard

Diagram

Description automatically generated

What is the Solution?

Solution 1: Use separate instruction and data memories

Solution 2: Allow memory to read and write more than one word per cycle

Solution 3: Stall

Diagram, engineering drawing, schematic

Description automatically generated

nop instruction performs nothing.

Control Hazard Solutions

Stall: Wait until decision is clear

It is possible to move up decision to 2nd stage by adding extra hardware to check registers as being read

Diagram

Description automatically generated

Predict: Guess one direction than back up if wrong

Predict not taken

Diagram

Description automatically generated

Redefine branch behaviour (takes place after next instruction) “delayed branch”

Put different instruction (an independent instruction) between branch and the next instruction.

Diagram

Description automatically generated

Data Hazard on r1

Problem: r1 cannot be read by othe instructions before it is written by the add.

* add r1, r2, r3
* sub r4, r1, r3
* and r6, r1, r7
* or r8, r1, r9
* xor r10, r1, r11

Diagram, schematic

Description automatically generatedDependencies backwards in time are hazards

sub is the most problematic because r1 is available after 2 cycles. If we can solve it we can solve others.

After here correct value r1 is at our datapath.

Forward result from one stage to another



Diagram, schematic

Description automatically generated

“or” instruction is OK if define read/write properly

A PIPELINED DATAPATH DESIGN

Pipelining the Load Instruction

Diagram

Description automatically generated

The 5 independent functional units in the pipeline datapath are:

* Instruction memory for the Ifetch stage
* Register File’s Read ports (bus A and bus B) for the Reg/Dec stage
* ALU for the Exec stage
* Data memory for the Mem stage
* Register File’s Write port (bus W) for the Wr stage

The 4 stages of R-type:

Diagram

Description automatically generated

* Ifetch: Instruction fetch
  + Fetch the instruction from the instruction memory
  + Update PC
* Reg/Dec: Registers fetch and instruction decode
* Exec:
  + ALU operates on the two register operands
* Wr: Write the ALU output back to the register file

Pipelining the R-type and Load Instruction

Diagram

Description automatically generated

We have pipeline confict or structural hazard:

* Two instructions try to write to the register file at the same time!
* Only one write port

Important Observation

Each functional unit can only be used once per instruction

Each functional unit must be used at the same stage for all instructions:

* Load uses Register File’s Write Port during its 5th stage

Diagram

Description automatically generated

* R-type uses Register File’s Write Port during its 4th stage

A picture containing text, device, meter

Description automatically generated

* 2 ways to solve this pipeline hazard.

**Solution 1: Insert “Bubble” into the Pipeline**

Diagram

Description automatically generated

Insert a “bubble” into the pipeline to prevent 2 writes at the same cycle

* The control logic can be complex
* Lose instruction fetch and issue opportunity

No instruction is started in Cycle 6!

**Solution 2: Delay R-type’s Write by One Cycle**

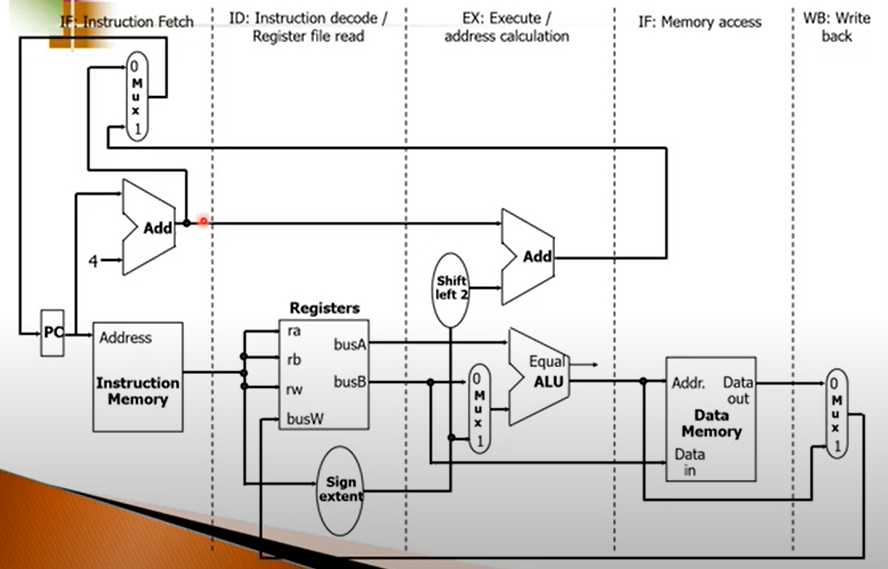
Delay R-type’s register write by one cycle:

* Now R-type instruction also use Reg File’s write port at Stage 5
* Graphical user interface

  Description automatically generatedMem stage is a NOOP stage: nothing is being done.

At each cycle, an instruction finishes its execution so in performance we don’t have much problem.

Single Cycle Datapath



In WB stage, we use register block for writing results. We don’t use it for reading anything. So register block can be used by ID stage and WB stage at the same time.

We must isolate each stage from others.

Pipelined Version of the Datapath

Diagram, schematic

Description automatically generated



Orange boxes: registers

We can write to registers at the end of cycle. So we are sure that content of a register will not change until the end of the cycle.

IF/ID registers preserve the values which belong to instruction 2 (i2).  
ID/EX registers preserve the values which belong to i3.

So for example, new values in red circled wires do not affect the register contents because they cannot be written to register until the end of the cycle.

At the end of the cycle, data belongs to i2 will be written to ID/EX register. Then we have i2 in EX stage instead of i3.

Her register, instructionın sonraki stagelerde ihtiyaç duyduğu datayı içinde saklıyor.

Eğer instruction lw veya sw değilse MEM stagei atlanır.

**IF: The first pipe stage of a load instruction**

Diagram, schematic

Description automatically generated

Write to IF/ID occurs at the end of the cycle.

Diagram

Description automatically generated**ID: The second pipe stage of a load instruction**



Assume we have 1 instruction. “Registers” may also be used for write at the same time.

PC + 4 is also written to ID/EX registers because it may be used if the instruction is branch. Whatever you want to use in the remaining stages, you have to carry them with you.

ID/EX can hold 138 bits.

**EX: The third pipe stage of a load instruction**

Diagram, schematic

Description automatically generated



Add is used for branch instructions.

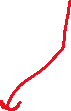
rt content for sw.

EX/MEM can hold 102 bits.

**MEM: The fourth pipe stage of a load instruction**

Diagram, schematic

Description automatically generated



If instruction is sw, left side of memory is painted.

If instruction is neither sw nor lw, memory is not painted. We follow this path.

**WB: The fifth pipe stage of a load instruction**

Diagram

Description automatically generated

İlk 2 cycle hep aynı, instructionın ne olduğu zaten 2. stagede ortaya çıkıyor.

**IF: The first pipe stage of a R-type instruction**

Diagram, schematic

Description automatically generated

**ID: The second pipe stage of a R-type instruction**

Diagram, schematic

Description automatically generated

**EX: The third pipe stage of a R-type instruction**

Diagram

Description automatically generated

**MEM: The fourth pipe stage of a R-type instruction**

Diagram, schematic

Description automatically generated

**WB: The fifth pipe stage of a R-type instruction**

Diagram

Description automatically generated

An Example to Clarify Pipelining

Since many instructions are simultaneously are executing in a single cycle datapath, it can be difficult to understand.

The following code will be examined:

* lw $10, 20($1)
* sub $11, $2, $3

Diagram

Description automatically generated

CLOCK 1

Diagram, schematic

Description automatically generated

Only machine code of the instruction is read from the memory.

Diagram, schematic

Description automatically generatedCLOCK 2

CLOCK 3

Diagram

Description automatically generated

Diagram, schematic

Description automatically generatedCLOCK 4

CLOCK 5

Diagram

Description automatically generated

Summary: Pipelining

What makes it easy

* all instructions are the same length
* just a few instruction formats
* memory operands appear only in loads and stores

What makes it hard

* structural hazards: suppose we had only one memory
* control hazards: need to worry about branch instructions
* data hazards: an instruction depends on a previous instruction

Pipelining is a fundamental concept

* multiple steps using distinct resources

The modern processors really makes it hard:

* exception handling (for example if lw tries to access invalid place in memory)
* trying to improve performance with out-of-order execution, etc.

A PIPELINED DATAPATH CONTROL

Pipelined Datapath with Control Signals

Diagram, schematic

Description automatically generated

In control part, we don’t only have control unit that produces the required control signals but also we will insert hazard units which are responsible from avoiding the hazards.

Green ones are the control signals that must be produced by control unit.

Each cycle, control signals are changed according to current instruction.

ALU Control is very similar to the previous ALU control. It gets ALUop from main control and it gets the function field and accordingly produces the required ALU select signal.

Table

Description automatically generated with low confidenceRecall: ALU Control Bits

The Values of Control Lines for The Last Three Pipeline Stages

A picture containing calendar

Description automatically generated

Control units only outputs signal to the first register which is ID/EX.

Different signals are required in different stages.

Diagram, schematic

Description automatically generated

opcode

Control unit kararı sadece opcode’a bakarak veriyor. Yani output sadece inputa (opcode) bağlı. O yüzden control unit combinational. Single cycle’daki control unitin aynısını kullanabiliriz.

Her cycle başka bir instruction geliyor, her cycle başka bir instructiona göre sinyal üretiyorum (ben control unitim). Yani senin sinyallerin kaybolur. O yüzden onları bir registera kaydedelim -> ID/EX. Her cycleda bu sinyalleri sonraki stage’e taşıyalım.

An Example to Clarify Pipelined Control

Let’s look at what’s happening in the pipeline for the following program:

* lw $10, 20($1)
* sub $11, $2, $3
* and $12, $4, $5
* or $13, $6, $7
* add $14, $8, $9

Code doesn’t have any data, control, or structural hazard.

Diagram, schematic

Description automatically generated

Diagram, schematic

Description automatically generated

Control unit produces all required control signals for lw.

Diagram, schematic

Description automatically generated

Diagram, schematic

Description automatically generated

Diagram, schematic

Description automatically generated

Structural hazard solution 🡪 Put multiple copies of same structure

Data Hazards

Previous example shows us how independent instructions that do not use the results calculated by prior instructions are executed.

This is not the case with real programs.

Let’s look at the following code sequence:

* sub $2, $1, $3
* and $12, $2, $5
* or $13, $6, $2
* add $14, $2, $2
* sw $15, 100($2)

The last four instructions are all dependent on the register $2 of the first instruction.

Assume that register $2 had the value of 10 before the subtract instruction and -20 afterwards.

Diagram

Description automatically generated

If the first half of the cycle, the write operation is finished and read operation must be finished at the second half of the cycle so that add works correctly.

Simple solution: Compiler inserts nop instructions between the sub-and instructions.

* nop (no operation) instruction neither modifies data nor writes a result.
  + sub $2, $1, $3
  + nop
  + nop
  + and $12, $2, $5
  + or $13, $6, $2
  + add $14, $2, $2
  + sw $15, 100($2)

Result: It works but 2 clock cycles will be wasted.

* Performance decrease

**Data Hazard Detection & Forwarding**

It is possible to detect data hazard and then forward the proper value to resolve the hazard.

When an instruction tries to read a register in its EX stage that an earlier instruction intends to write in its WB stage.

This is the case between sub-and instruction below:



Diagram

Description automatically generated



This hazard can be detected by simply checking:

* EX/MEM.RegisterRd = ID/EX.RegisterRs = $2
* Check if destination of the instruction at memory stage ($2) in EX/MEM register is equal to register rs value ($2) at ID/EX register. Also we should check for rt register.

Another hazard is between sub-or instructions:

Diagram

Description automatically generated

This hazard can be detected by simply checking:

* MEM/WB.RegisterRd = ID/EX.RegisterRt = $2

There is no data hazard between sub-add and sub-sw instructions.

**Summary of Data Hazard Conditions**

Text

Description automatically generated

If previous instruction is not writing to register, there will be no hazard. This must be checked by the forwarding unit too. This is why we have RegWr signals. In this situation, directly select the direct contents of register inputs.

The Pipelined Datapath with Forwarding

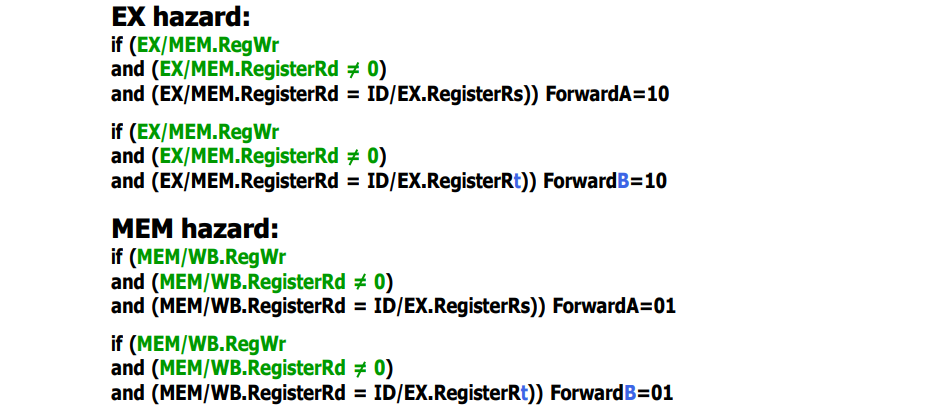
Diagram, schematic

Description automatically generated

Another problem: What happens if $0 is used as a destination register?

* A non-zero value would not be forwarded
* 0 numaralı registera yazmak istese de değiştiremeyeceğini biliyoruz.

Therefore, hazard detection should be the following:



EX hazard 🡪 arka arkaya gelen 2 instruction için (biri EX’te diğeri MEM’de)

MEM hazard 🡪 EX’te ve WB’de olan 2 instruction arası hazard

Data hazard olması için 3 koşul:

* MEM stageindeki instruction register değiştiren bir instruction olmalı
* Değiştirmek istenilen register adresi 0 olmamalı
* Değiştirmek istenilen register adresiyle kendisinden bir veya iki sonraki instructionın (EX stageindeki) okumak istediği register adresi aynı olmalı

Diagram

Description automatically generated

Bu tarz bir durumda iki if statement da (1. ve 3.) doğru olur. Öncelik vermemiz gereken add ve add arası hazard çünkü en güncel data, bana en yakın data o. Bunun için şu şartı ekleriz:

A screenshot of a computer

Description automatically generated with medium confidence

lw instructionının registera yazmak istediği değer ALU outputu değil. Yani EX stageinde “or $3, $2, …” ve MEM stageinde de “lw $2, …” olsun. ALU outputu registera yazmak istediğimiz değer olmadığı için henüz $2 hazır değil. Data memorynin çıkışından da alamaz çünkü zaten değer cycle sonunda çıkacak. Bu değeri ALU’ya yetiştirip, ALU’nun bu sonuçla işlem yapıp sonuç hesaplaması cycle süremizi uzatır.

Diagram

Description automatically generated

Kırmızı okla yani gelecekten geçmişe forwarding yapamazsın. 1 cycle bekleyeceksin ki $2 hazır olsun. Cycle süremiz 2 katına çıkar, bütün instructionlar yavaşlar.

Bu durumda lw ve and arasına nop koyarız.

Şunlara bakacağız:

* EX stageinde lw var mı?
* Bu lw’ün yazmak istediği register ID stageindeki instruction okumak istiyor mu?

Hazard Detection

The control for the hazard detection unit is:

if (ID/EX.MemRd ----------------------> Check if the instruction is a load

and

((ID/EX.RegisterRt = IF/ID.RegisterRs) or

(ID/EX.RegisterRt = IF/ID.RegisterRt)))

stall the pipeline

Diagram

Description automatically generated

The Pipelined Datapath with Forwarding and Hazard Detection Unit

Diagram, schematic

Description automatically generated

,

MUX’a 0 verdik (nop için) ki bütün control signaller 0 olsun ve instruction hiçbir şey yapmasın. Yani lw ile sonraki instruction arasına nop giriyor.

lw’ten sonra or gelsin ve hazard olsun. Araya nop soktuk. Or’un datası ID/EX registerına yazılamadı. Yeni instruction geldiği için IF/ID registerı da doldu. Or’un datasını kaybetmemek için hazard detection unit aynı zamanda IF/ID’ye bir şey yazılmasına da engel olur (IF/IDWrite’a 0 vererek).

Yeni instructionın IF/ID’ye yazılmasını engelledik. Bu zaman da PC 4 artacak ve instructionın datasını kaybedeceğiz. Yeni gelen instructionın datasını kaybetmemek için hazard detection unit PCWrite’a 0 vererek PC’ye PC+4 yazılmaz. Böylece PC halen yeni gelen instructionın adresini gösterir.

lw ve or arasında zaten halihazırda başka bir instruction olsaydı bunu forwarding unit çözebilirdi.

Forwarding unit combinational. Tamamen XORlardan ANDlerden falan oluşuyor. Hazard detection unit de aynı şekilde.

Branch Hazards

Diagram, schematic

Description automatically generated

Branch makes the decision in MEM stage.

Datapath for Branch (including HW to flush the pipeline)

Diagram, schematic

Description automatically generated



Comparator next to “Registers” checks equality of rs and rt register contents. If they are equal, branch makes the jump decision at ID stage. Equality of rs and rt are always checked even if instruction is branch or not. This is an additional operation, not much problem.

Next instruction is in the IF stage. If you make the jump decision, only instruction we should kill is that next instruction. We only lose 1 cycle.

Branch address is computed in red circled area.

At the end of the cycle, next instruction content is written in IF/ID register. In order to avoid that, we need to delete all data related to that next instruction. Instead of 32 bit of the machine code instruction, now we will have 32-bit of 0s (nop). After that nop instruction, the instruction that we need to jump comes.

If it doesn’t branch, then we don’t need to kill that next instruction. We don’t need to put any additional nop instruction in between.

We always predict not to branch. If we don’t branch, we lose no cycles. If we branch, we only lose 1 cycle.

**Example: Branch Taken**

Diagram

Description automatically generated

If branch is taken IF.Flush will be made 1 so and instruction will not be able to write its data to IF/ID register.

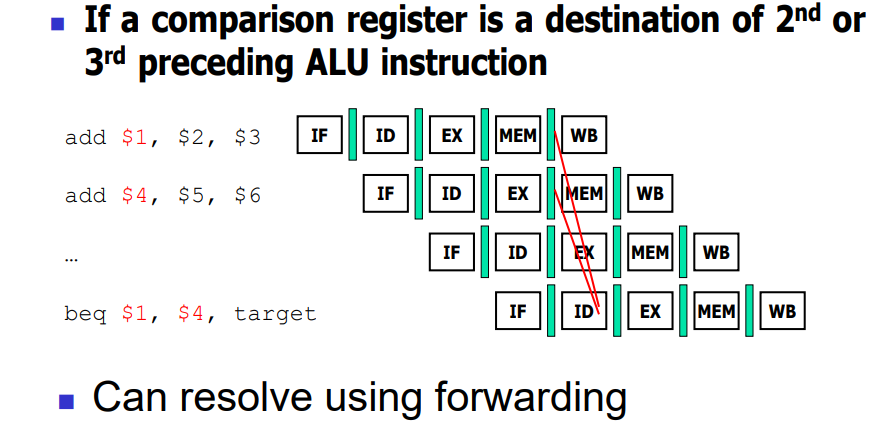
Diagram, schematic

Description automatically generated

lw is actual instruction at the branch location.

Assume add address is 44. At the end of the cycle, instead of 48 (next instruction after and), PC value will be 72 (44 + 7\*4).

Data Hazards for Branches



Diagram

Description automatically generated



Content of $4 is not ready at. It will be ready at the end of the cycle. We cant perform forwarding shown with red line.

Actually you need to put forwarding unit because this new forwarding unit will be responsible to solve hazard between ID and MEM stage. Also you have to put nop instruction in between previous instruction and beq (in EX stage).

Diagram

Description automatically generated

Diagram

Description automatically generated

lw’ün yazmak istediğini branch okumak istiyorsa araya 2 cycle birden koymak lazım çünkü lw datası MEM stageinden sonra ortaya çıkacak.