Pipelining

Pipelining

- **Pipelining** is an implementation technique where multiple instructions are overlapped in execution. The computer pipeline is divided in stages. Each stage completes a part of an instruction in parallel. The stages are connected one to the next to form a pipe - instructions enter at one end, progress through the stages, and exit at the other end.

- Pipelining does not decrease the time for individual instruction execution. Instead, it increases instruction throughput.
  The **throughput** of the instruction pipeline is determined by how often an instruction exits the pipeline.

Classification of Instruction Sets

- The instruction sets can be differentiated by:
  - Operand storage in the CPU
  - Number of explicit operands per instruction
  - Operand location
  - Operations
  - Type and size of operands

- The type of internal storage in the CPU is the most basic differentiation. Types:
  - **a stack** (the operands are implicitly on top of the stack)
  - **an accumulator**
  - **a set of registers** (all operands are explicit either registers or memory locations)

Classification of General Purpose Register Machines

There are two major instruction set characteristics that divide GPR architectures. They concern

- whether an ALU instruction has two or three operands

ADD R3, R1, R2 — R3 <-R1 + R2

or

ADD R1, R2 — R1 <- R1 + R2

- how many of the operands may be memory addressed in ALU instruction
  - Register – Register (Load/ Store)

  ADD R3, R1, R2 (R3 <- R1 + R2)
  - Register – Memory

  ADD R1, A (R1 <- R1 + A)
  - Memory – Memory

  ADD C, A, B (C <- A+B)

Addressing Modes

Addressing modes	**Example Instruction**	**Meaning**	**When used**
Register	Add R4,R3	R4 <- R4 + R3	When a value is in a register
Immediate	Add R4, #3	R4 <- R4 + 3	For constants
Displacement	Add R4, 100(R1)	R4 <- R4 + M[100+R1]	Accessing local variables
Register deffered	Add R4,(R1)	R4 <- R4 + M[R1]	Accessing using a pointer or a computed address
Indexed	Add R3, (R1 + R2)	R3 <- R3 + M[R1+R2]	Useful in array addressing: R1 - base of array R2 - index amount
Direct	Add R1, (1001)	R1 <- R1 + M[1001]	Useful in accessing static data
Memory deferred	Add R1, @(R3)	R1 <- R1 + M[M[R3]]	If R3 is the address of a pointer *p*, then mode yields **p*
Auto-increment	Add R1, (R2)+	R1 <- R1 +M[R2] R2 <- R2 + *d*	Useful for stepping through arrays in a loop. R2 - start of array *d* - size of an element
Auto-decrement	Add R1,-(R2)	R2 <-R2-*d* R1 <- R1 + M[R2]	Same as autoincrement. Both can also be used to implement a stack as push and pop
Scaled	Add R1, 100(R2)[R3]	R1<-R1+M[100+R2+R3**d*]	Used to index arrays. May be applied to any base addressing mode in some machines.

Interpreting Memory Addresses

There are two different conventions for ordering the bytes within a word:

- Little Endian (followed by DEC and Intel)
- Big Endian (followed by IBM, Motorola and others)

- When operating within one machine, the byte order is often unnoticeable - only programs that access the same locations as both words and bytes can notice the difference. However, byte order is a problem when exchanging data among machines with different ordering.

A simple Implementation of a RISC Instruction Set

1. *Instruction fetch cycle (IF):*

Send the program counter (PC) to memory and fetch the current instruction from memory. Update the PC to the next sequential PC by adding 4 (since each instruction is 4 bytes) to the PC.

2. *Instruction decode/register fetch cycle (ID):*

Decode the instruction and read the registers corresponding to register source specifiers from the register file. Do the equality test on the registers as they are read, for a possible branch. Sign-extend the offset field of the instruction in case it is needed. Compute the possible branch target address by adding the sign-extended offset to the incremented PC. In an aggressive implementation, which we explore later, the branch can be completed at the end of this stage by storing the branch-target address into the PC, if the condition test yielded true.

Decoding is done in parallel with reading registers, which is possible because the register specifiers are at a fixed location in a RISC architecture. This technique is known as fixed-field decoding. Note that we may read a register we don’t use, which doesn’t help but also doesn’t hurt performance. (It does waste energy to read an unneeded register, and power-sensitive designs might avoid this.) Because the immediate portion of an instructionis also located in an identical place, the sign-extended immediate is also calculated during this cycle in case it is needed.

A simple Implementation of a RISC Instruction Set

3. Execution/effective address cycle (EX):

The ALU operates on the operands prepared in the prior cycle, performing one of three functions depending on the instruction type.

■ Memory reference—The ALU adds the base register and the offset to form the effective address.

■ Register-Register ALU instruction—The ALU performs the operation specified by the ALU opcode on the values read from the register file.

■ Register-Immediate ALU instruction—The ALU performs the operation specified by the ALU opcode on the first value read from the register file and the sign-extended immediate.

In a load-store architecture the effective address and execution cycles can be combined into a single clock cycle, since no instruction needs to simultaneously calculate a data address and perform an operation on the data.

A simple Implementation of a RISC Instruction Set

- 4. *Memory access (MEM):*

If the instruction is a load, the memory does a read using the effective address computed in the previous cycle. If it is a store, then the memory writes the data from the second register read from the register file using the effective address.

- 5. *Write-back cycle (WB):*

Register-Register ALU instruction or load instruction:

Write the result into the register file, whether it comes from the memory system (for a load) or from the ALU (for an ALU instruction)

Pipeline Hazards

- There are situations, called hazards, that prevent the next instruction in the instruction stream from being executing during its designated clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining.
- There are three classes of hazards:
    1. Structural Hazards. They arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.
    2. Data Hazards. They arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
    3. Control Hazards.They arise from the pipelining of branches and other instructions that change the PC.

Structural Hazards

A machine has shared a single-memory pipeline for data and instructions. As a result, when an instruction contains a data-memory reference(load), it will conflict with the instruction reference for a later instruction (instr 3):

- Clock cycle number
- Instr 1 2 3 4 5 6 7 8
- Load IF ID EX MEM WB
- Instr 1 IF ID EX MEM WB
- Instr 2 IF ID EX MEM WB
- Instr 3 IF ID EX MEM WB

Structural hazards

To resolve this, we stall the pipeline for one clock cycle when a data-memory access occurs. The effect of the stall is actually to occupy the resources for that instruction slot. The following table shows how the stalls are actually implemented.

- Clock cycle number

Instr	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Load	IF	ID	EX	MEM	WB				
Instr 1		IF	ID	EX	MEM	WB			
Instr 2			IF	ID	EX	MEM	WB		
Stall				**bubble**	**bubble**	**bubble**	**bubble**	**bubble**	
Instr 3					IF	ID	EX	MEM	WB

- Instruction 1 assumed not to be data-memory reference (load or store), otherwise Instruction 3 cannot start execution for the same reason as above.

- To simplify the picture it is also commonly shown like this:
- Clock cycle number

Instr	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Load	IF	ID	EX	MEM	WB				
Instr 1		IF	ID	EX	MEM	WB			
Instr 2			IF	ID	EX	MEM	WB		
Instr 3				**stall**	IF	ID	EX	MEM	WB

Data Hazards

A major effect of pipelining is to change the relative timing of instructions by overlapping their execution. This introduces data and control hazards. **Data hazards** occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on the unpipelined machine.
Consider the pipelined execution of these instructions:

		1	2	3	4	5	6	7	8	9
ADD	**R1**, R2, R3	IF	ID	EX	MEM	**WB**				
SUB	R4, R5, **R1**		IF	**ID$_{sub}$**	EX	MEM	WB			
AND	R6, **R1,** R7			IF	**ID$_{and}$**	EX	MEM	WB		
OR	R8, **R1**, R9				IF	**ID$_{or}$**	EX	MEM	WB	
XOR	R10,**R1**,R11					IF	**ID$_{xor}$**	EX	MEM	WB

Data hazards classification

- ***RAW (read after write)*** *- **j** tries to read a source before **i** writes it, so **j** incorrectly gets the old value.*This is the most common type of hazard and the kind that we use forwarding to overcome.
- ***WAW (write after write)*** *- **j** tries to write an operand before it is written by **i**. The writes end up being performed in the wrong order, leaving the value written by **i** rather than the value written by **j** in the destination.*

This hazard is present only in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled. The DLX integer pipeline writes a register only in WB and avoids this class of hazards.

- ***WAR (write after read)*** *- **j** tries to write a destination before it is read by **i** , so **i** incorrectly gets the new value.*

This can not happen in our example pipeline because all reads are early (in ID) and all writes are late (in WB). This hazard occurs when there are some instructions that write results early in the instruction pipeline, and other instructions that read a source late in the pipeline.

Forwarding

The problem with data hazards, introduced by this sequence of instructions can be solved with a simple hardware technique called ***forwarding***.

The key insight in forwarding is that the result is not really needed by SUB until after the ADD actually produces it. The only problem is to make it available for SUB when it needs it.

		1	2	3	4	5	6	7
ADD	**R1**, R2, R3	IF	ID	EX	MEM	**WB**		
SUB	R4, R5, **R1**		IF	**ID$_{sub}$**	EX	MEM	WB	
AND	R6, **R1,** R7			IF	**ID$_{and}$**	EX	MEM	WB

Forwarding

If the result can be moved from where the ADD produces it (EX/MEM register), to where the SUB needs it (ALU input latch), then the need for a stall can be avoided.

Using this observation , forwarding works as follows:

- The ALU result from the EX/MEM register is always ***fed back*** to the ALU input latches.
- If the forwarding hardware detects that the previous ALU operation has written the register corresponding to the source for the current ALU operation, ***control logic*** selects the forwarded result as the ALU input rather than the value read from the register file.

Without forwarding our example will execute correctly with stalls:

		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
ADD	**R1**, R2, R3	IF	ID	EX	MEM	**WB**				
SUB	R4, R5, **R1**		IF	**stall**	**stall**	**ID$_{sub}$**	EX	MEM	WB	
AND	R6, **R1,** R7			**stall**	**stall**	IF	**ID$_{and}$**	EX	MEM	WB

As our example shows, we need to forward results not only from the immediately previous instruction, but possibly from an instruction that started three cycles earlier. Forwarding can be arranged from MEM/WB latch to ALU input also. Using those forwarding paths the code sequence can be executed without stalls:

		1	**2**	**3**	**4**	**5**	**6**	**7**
ADD	**R1**, R2, R3	IF	ID	**EX$_{add}$**	**MEM$_{add}$**	WB		
SUB	R4, R5, **R1**		IF	ID	**EX$_{sub}$**	MEM	WB	
AND	R6, **R1,** R7			IF	ID	**EX$_{and}$**	MEM	WB

The first forwarding is for value of **R1** from **EX$_{add}$** to **EX$_{sub}$** .
The second forwarding is also for value of **R1** from **MEM$_{add}$** to **EX$_{and}$**.
This code now can be executed without stalls.

Forwarding

To prevent a stall in this example, we would need to forward the values of R1 and R4 from the pipeline registers to the ALU and data memory inputs.

Stores require an operand during MEM, and forwarding of that operand is shown here.
The first forwarding is for value of **R1** from $\mathbf{EX_{add}}$ to $\mathbf{EX_{lw}}$.
The second forwarding is also for value of **R1** from $\mathbf{MEM_{add}}$ to $\mathbf{EX_{sw}}$.
The third forwarding is for value of **R4 from** $\mathbf{MEM_{lw}}$ to $\mathbf{MEM_{sw}}$.

		1	2	3	4	5	6	7
ADD	**R1**, R2, R3	IF	ID	$\mathbf{EX_{add}}$	$\mathbf{MEM_{add}}$	WB		
LW	**R4, d (R1)**		IF	ID	$\mathbf{EX_{lw}}$	$\mathbf{MEM_{lw}}$	WB	
SW	**R4,12(R1)**			IF	ID	$\mathbf{EX_{sw}}$	$\mathbf{MEM_{sw}}$	WB

Control Hazards