Chapter 5

Pipelining

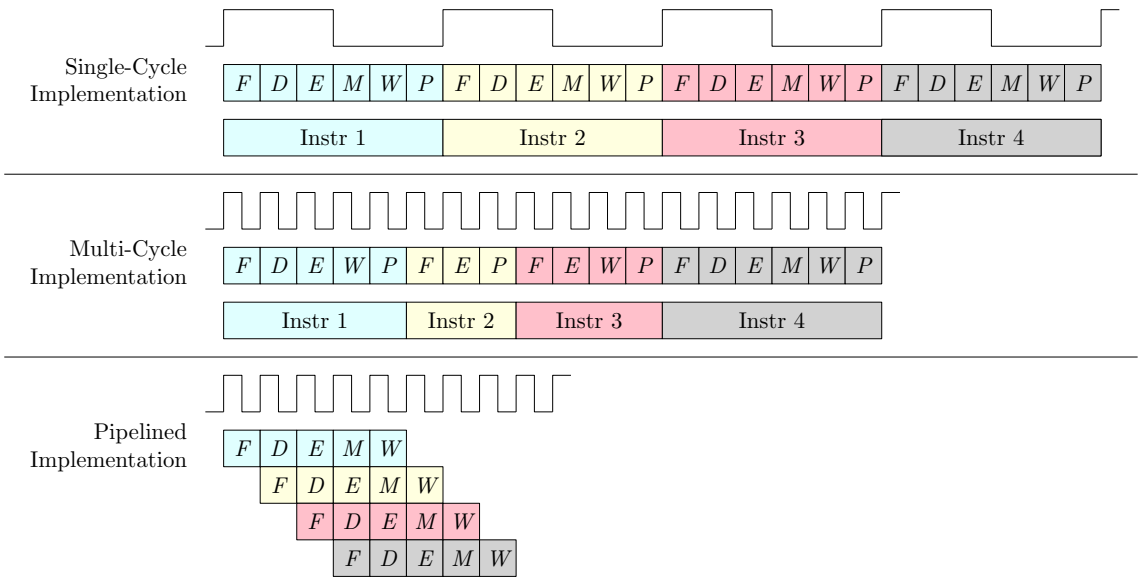
5.1 Sequential Implementation

From the multi-cycle implementation, only one computation stage is performed in one cycle. Therefore, a single instruction gets executed at a time. This type of implementation can be called as "*sequential implememtation*".

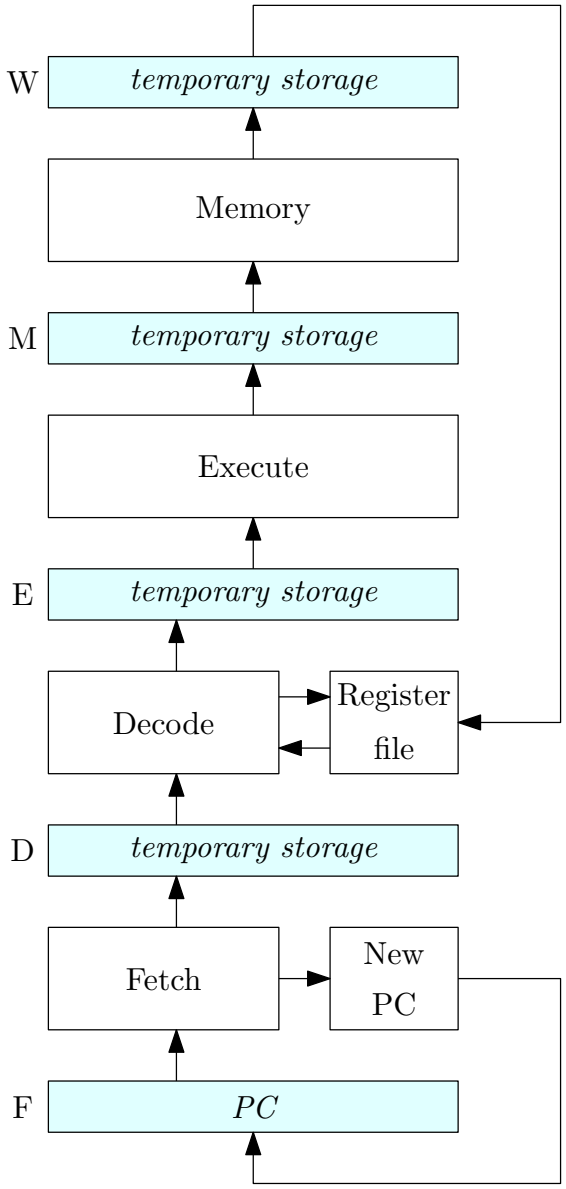
However, it can be easily seen that many parts of the datapath are still available in each cycle of the multi-cycle implementation. For example, when one instruction is decoded, the datapath for the *fetch* stage is available. It would improve the execution speed if we can control the *fetch* state to work on the next instruction.

5.2 Pipelining

Pipelining [BO10] is an implementation technique that utilizes all the parts of the datapath in one cycle. Therefore, multiple instructions executed in parallel. But the instructions are still executed according the program.

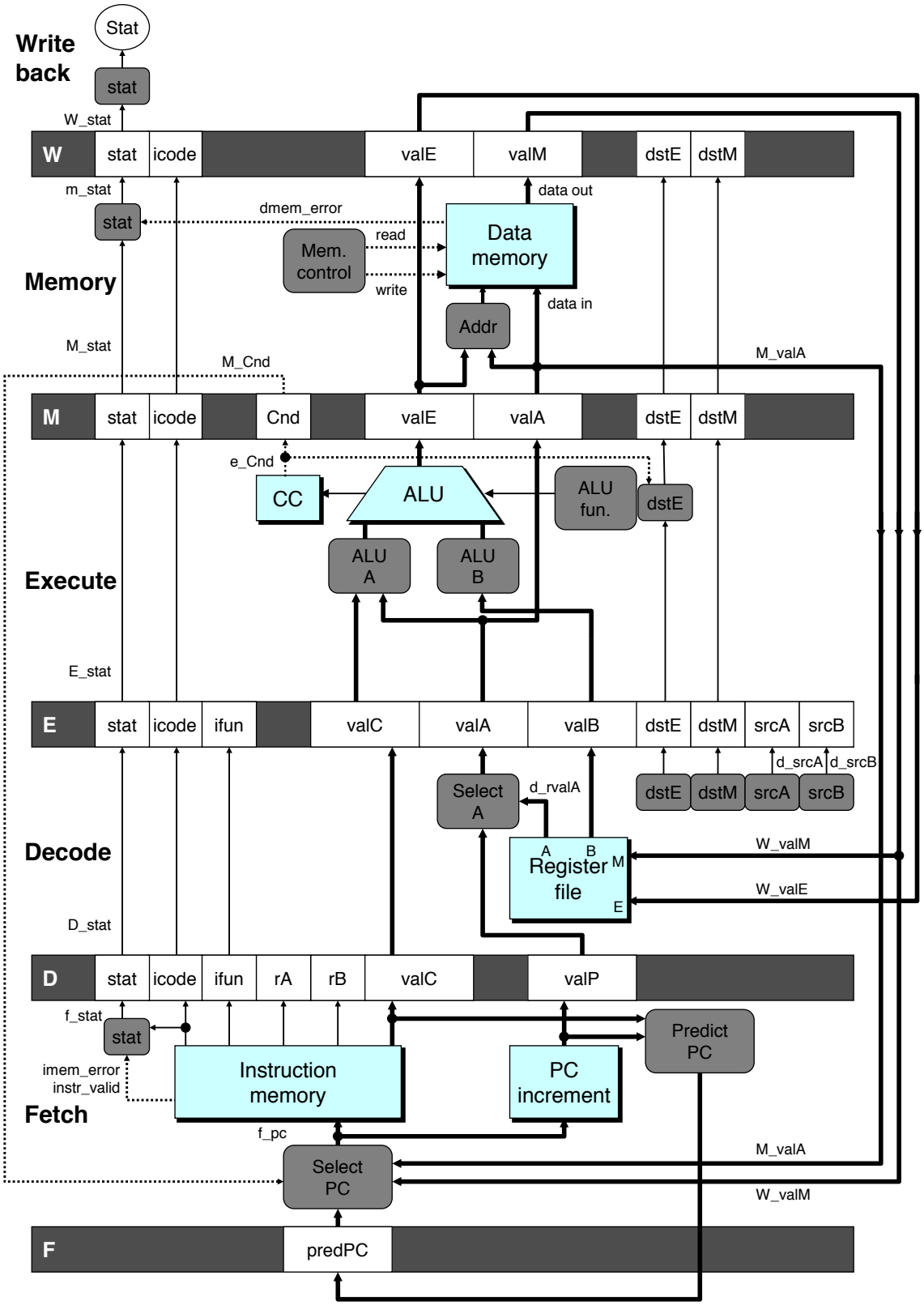


5.3 Datapath for the Pipelined Implementation



Each blue box represent a *temporary storage* for keeping the control signals and values passing from one stage to another stage. Adding these storages allows multiple instructions to be executed in the same datapath.

The *PC Update* stage has been removed since the address of the next instruction has to be computed and used immediately.


Stat
Write back
stat
W_stat
W
stat
icode
valE
valM
dstE
dstM
m_stat
data out
dmem_error
stat
read
Mem. control
Data memory
write
Memory
data in
Addr
M_stat
M_Cnd
M_valA
M
stat
icode
Cnd
valE
valA
dstE
dstM
e_Cnd
CC
ALU
ALU fun.
dstE
ALU A
ALU B
Execute
E_stat
E
stat
icode
ifun
valC
valA
valB
dstE
dstM
srcA
srcB
d_srcA
d_srcB
Select A
d_rvalA
dstE
dstM
srcA
srcB
W_valM
A
B
Register file
M
E
W_valE
Decode
D_stat
D
stat
icode
ifun
rA
rB
valC
valP
f_stat
stat
Predict PC
imem_error
instr_valid
Instruction memory
PC increment
Fetch
f_pc
M_valA
Select PC
W_valM
F
predPC

Exercise 5.1 Show how the following program is executed in the pipelined implementation of Y86 architecture?

```
i1) add    %rax, %rcx
i2) add    %rdx, %rdx
i3) irmov  $10, %r8
```

	1	2	3	4	5	6	7	8	9	10
i1										
i2										
i3										

Exercise 5.2 How many cycles required to execute the following assembly problem in *multi-cycle* and *pipelined* implementation? What is the speed-up we can obtain from the pipelined implementation?

```
i1) push   %rbx
i2) push   %r12
i3) irmov  $0, %rax
i4) add    %rcx, %rsi
i5) pop    %r12
i6) pop    %rbx
```

	1	2	3	4	5	6	7	8	9	10
i1										
i2										
i3										
i4										
i5										
i6										

Exercise 5.3 Let `%rax=10`, `%rcx=5`, and `%rdx=8`. What is the values of `%rax`, `%rcx` and `%rdx` after the following program ends?

```
add    %rax, %rcx
sub    %rcx, %rdx
```

5.4 Data Hazards

The output of an instruction may be used as an operand for another instruction. This is called "*data dependency*".

Data Hazard is a situation when the implementation changes the order of operand accesses so that it is different from the order when the instructions are sequentially executed. It can occur from three situations:

1. Read After Write (RAW)

```
i1) add  %rcx, %rax     # %rax = %rax + %rcx
i2) add  %rax, %rdx     # %rdx = %rdx + %rax
```

```
i1) push %rcx
i2) push %rdx
```

2. Write After Read (WAR)

```
i1) add  %rax, %rcx
i2) add  %rdx, %rax
```

3. Write After Write (WAW)

```
i1) add  %rcx, %rax
i2) add  %rdx, %rax
```

Exercise 5.4 Mark the data hazards in the following sequence of instructions.

```
i1) irmov  $10, %rax
i2) rrmov  %rax, %rbx
i3) mrmov  4(%rax), %rcx
i4) add    %rcx, %rdx
i5) push   %rdx
i6) pop    %rax
```

The data hazards may be resolved by *stalling the pipeline*. This is performed by making the control unit hold back a number of instructions in the pipeline until the condition for the hazard no longer holds.

Typically, the hazard condition can be detected by the control until in the *Decode* stage.

Exercise 5.5 Show how the following program is *properly* executed in the pipelined implementation.

```
i1) irmov $10, %rax
i2) rrmov %rax, %rbx
i3) mrmov 4(%rax), %rcx
```

	1	2	3	4	5	6	7	8	9	10
i1										
i2										
i3										

Exercise 5.6 Show how the following program is executed properly in the pipelined implementation.

```
i1) irmov $1, %rcx
i2) irmov $2, %rdx
i3) add   %rdx, %rcx
i4) mrmov 4(%rcx), %rax
i5) xor   %rax, %rdx
i6) rrmov %rdx, %rbp
i7) push  %rbp
```

	1	2	3	4	5	6	7	8	9	10
i1										
i2										
i3										
i4										
i5										
i6										
i7										

	11	12	13	14	15	16	17	18	19	20
i1										
i2										
i3										
i4										
i5										
i6										
i7										

	21	22	23	24	25	26	27	28	29	30
i1										
i2										
i3										
i4										
i5										
i6										
i7										

Exercise 5.7 From the following program,can we reduce the number of cycles by exchanging the order of instructions but the program still yields the same output?

```
i1) irmov $5, %rax
i2) irmov $10, %rcx
i3) add   %rax, %rcx
i4) irmov $20, %rdx
i5) add   %rdx, %rsi
i6) push  %rbp
i7) push  %rdi
```

Before exchanging the order

	1	2	3	4	5	6	7	8	9	10
i1										
i2										
i3										
i4										
i5										
i6										
i7										

	11	12	13	14	15	16	17	18	19	20
i1										
i2										
i3										
i4										
i5										
i6										
i7										

	21	22	23	24	25	26	27	28	29	30
i1										
i2										
i3										
i4										
i5										
i6										
i7										

After exchanging the order

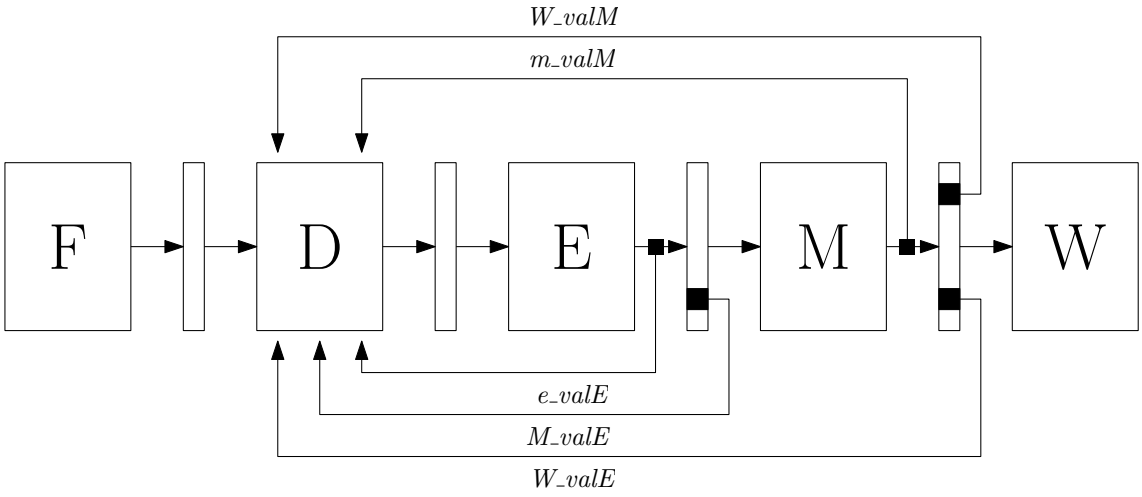
	1	2	3	4	5	6	7	8	9	10

	11	12	13	14	15	16	17	18	19	20

	21	22	23	24	25	26	27	28	29	30

5.4.1 Data Forwarding

Data Forwarding or *forwarding* is a technique to resolve the data hazards by passing a value directly from one stage to an earlier stage in order to avoid stalling.

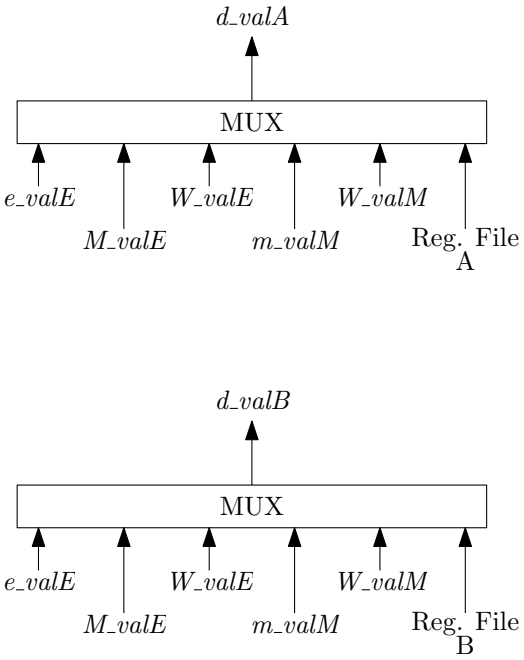


Here are the values that we forward to both *valA* and *valB* in the *Decode* stage:

1. current value of *valE* (*e_valE*)
2. *valE* from the last cycle (*M_valE*)
3. *valE* from the one before last (*W_valE*)
4. current value of *valM* (*m_valM*)
5. *valM* from the last cycle (*W_valM*)

Note: We use a small letter prefix to denote the current value e.g. *e_valE*, *m_valM*. We use a capital letter prefix to denote the value stored in the temporary storage.

Here are how the values forwarded,



Note: *d_valA* and *d_valB* are the value of *valA* and *valB* in the *Decode* stage.

Example 5.1 Show how the data forwarding helps avoid stalling.

```
i1) irmov $5, %rax
i2) irmov $10, %rcx
i3) add   %rax, %rcx
```

	1	2	3	4	5	6	7	8	9	10	*valA*	*valB*
i1												
i2												
i3												

```
i1) irmov $5, %rax
i2) irmov $10, %rcx
i3) nop
i4) add   %rax, %rcx
```

	1	2	3	4	5	6	7	8	9	10	*valA*	*valB*
i1												
i2												
i3												

```
i1) irmov $10, %rcx
i2) mrmov 5(%rbx), %rax
i3) add   %rcx, %rax
```

	1	2	3	4	5	6	7	8	9	10	*valA*	*valB*
i1												
i2												
i3												

```
i1) mrmov 5(%rbx), %rax
i2) irmov $10, %rcx
i3) add   %rcx, %rax
```

	1	2	3	4	5	6	7	8	9	10	*valA*	*valB*
i1												
i2												
i3												

Exercise 5.8 Show how the following program is executed in the pipelined implementation with Data Forwarding?

```
i1) irmov $1, %rcx
i2) irmov $2, %rdx
i3) add   %rdx, %rcx
i4) mrmov 4(%rcx), %rax
i5) xor   %rax, %rdx
i6) rrmov %rdx, %rbp
i7) push  %rbp
```

	1	2	3	4	5	6	7	8	9	10	*valA*	*valB*
i1												
i2												
i3												
i4												
i5												
i6												
i7												

	11	12	13	14	15	16	17	18	19	20
i1										
i2										
i3										
i4										
i5										
i6										
i7										

5.5 Control Hazards

A conditional jump instruction sets the next value of *PC* to either *valC* or *valP* according to the condition. This cannot be determined until the jump instruction has finished the *Execute* stage. This problem is called '*control hazard*'.

To solve the control hazards, a number of techniques have been proposed:

1. **Stall pipeline** clears the pipeline until the branch target is determined.
2. **Predict Branch Not Taken** always loads the next instruction into the pipeline. Then, the instructions can be discarded when the jump is taken.
3. **Dynamic Branch Prediction** uses a table indexed by the lower bits of the jump instruction address. This table contains a one-bit value indicating if the jump was previously taken.

Branch Instruction Address (binary)	Not Taken (0) or Taken (1)
0000	0
0001	0
0010	1
...	...
1111	0

Then, the branch prediction is based on the previous decision.

The following table shows a program with a control hazard when the jump is taken:

Address (decimal)	Instruction
0	sub %rax, %rcx
2	je 21
11	irmov 10, %rax
21	add %rdx, %rcx

Exercise 5.9 Show how the following program is executed using *Stall pipeline*?

Address	Instruction
0x00	irmov 0, %rax
0x0A	irmov 10, %rcx
0x14	irmov 1, %rdx
0x1E	cmp %rcx, %rax
0x21	je 0x35
0x2A	add %rdx, %rax
0x2C	jmp 0x1E
0x35	hlt

Exercise 5.10 Show how the following program is executed using *Predict Branch Not Taken*?

Address	Instruction
0x00	irmov 0, %rax
0x0A	irmov 10, %rcx
0x14	irmov 1, %rdx
0x1E	cmp %rcx, %rax
0x21	je 0x35
0x2A	add %rdx, %rax
0x2C	jmp 0x1E
0x35	hlt

Exercise 5.11 Show how the following program is executed using *Dynamic Branch Prediction using 4 lower bits*?

Address	Instruction
0x00	irmov 0, %rax
0x0A	irmov 10, %rcx
0x14	irmov 1, %rdx
0x1E	cmp %rcx, %rax
0x21	je 0x35
0x2A	add %rdx, %rax
0x2C	jmp 0x1E
0x35	hlt

References

[BO10] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective (2nd edition)*. Addison-Wesley, 2010.