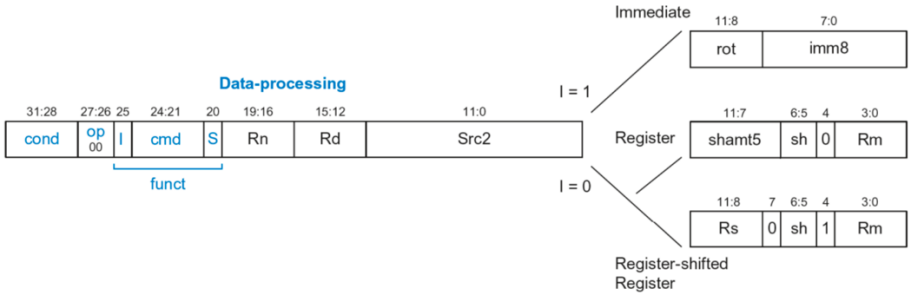
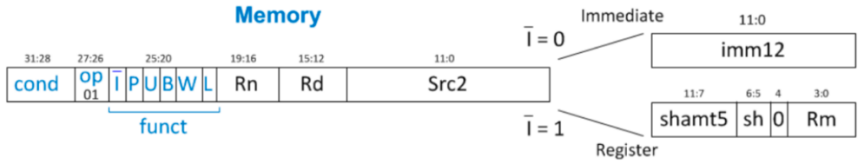
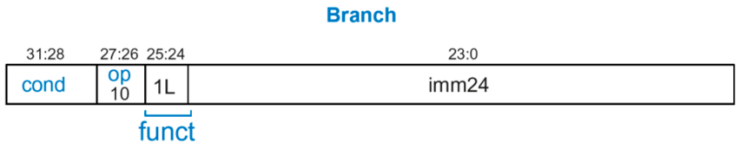
Mnemonic	Name		Operation
ADD	Addition	add Rd,Rn,Rm	Rd← Rn + (Rm *sh* shamt5)
SUB	Subtraction	sub Rd,Rn,Rm	Rd← Rn - (Rm *sh* shamt5)
AND	Bitwise And	and Rd,Rn,Rm	Rd← Rn & (Rm *sh* shamt5)
ORR	Bitwise Or	orr Rd,Rn,Rm	Rd← Rn | (Rm *sh* shamt5)
MOV	Move to Register	mov Rd,Rm	Rd← (Rm *sh* shamt5)
STR	Store	str Rd,[Rn,imm12]	Mem[Rn + imm12] ← Rd
LDR	Load	ldr Rd,[Rn,imm12]	Rd ← Mem[Rn + imm12]
CMP	Compare	cmp Rd,Rn,Rm	set the flag if (Rn - Rm =0)
B	Branch	b imm24	PC ← (PC + 8) + (imm24<< 2)
BEQ	Branch if Equal	beq imm24	PC ← (PC + 8) + (imm24<< 2) if flag = 1
BL	Branch with Link	bl imm24	PC ← (PC + 8) + (imm24<< 2), R14 ← PC
BX	Branch and Exchange	bx Rm	PC ← Rm







Datapath Design

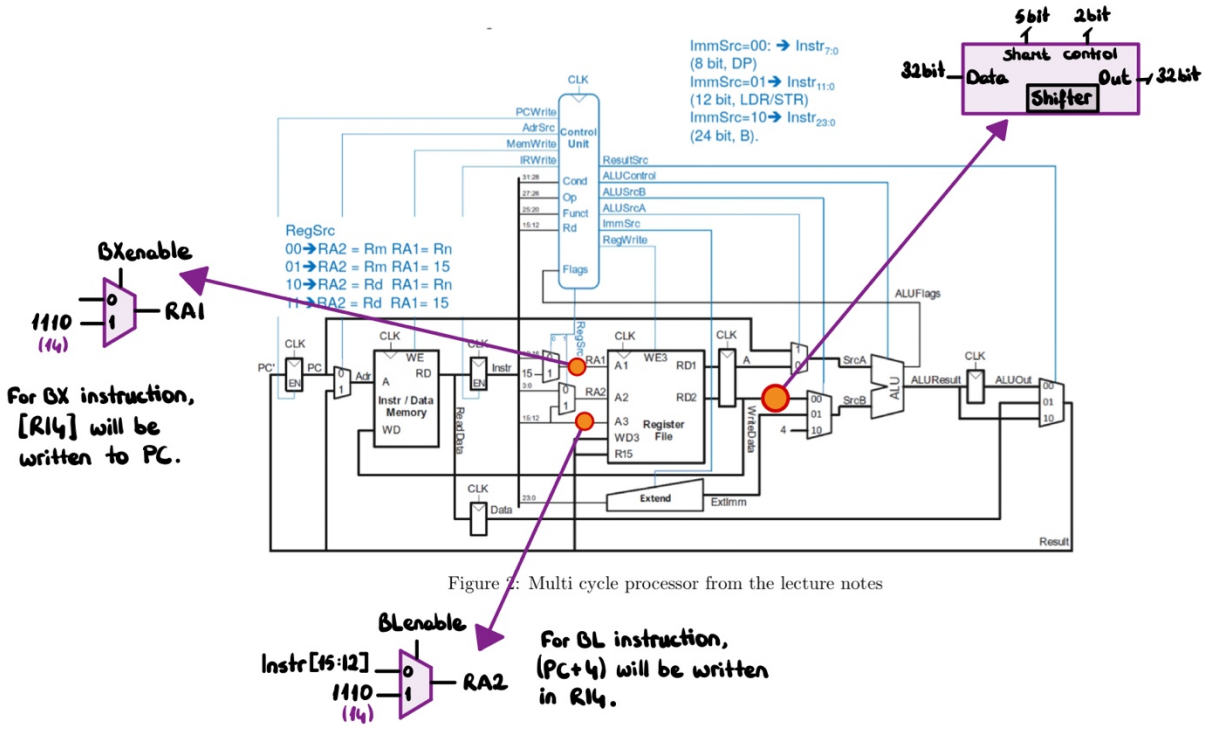
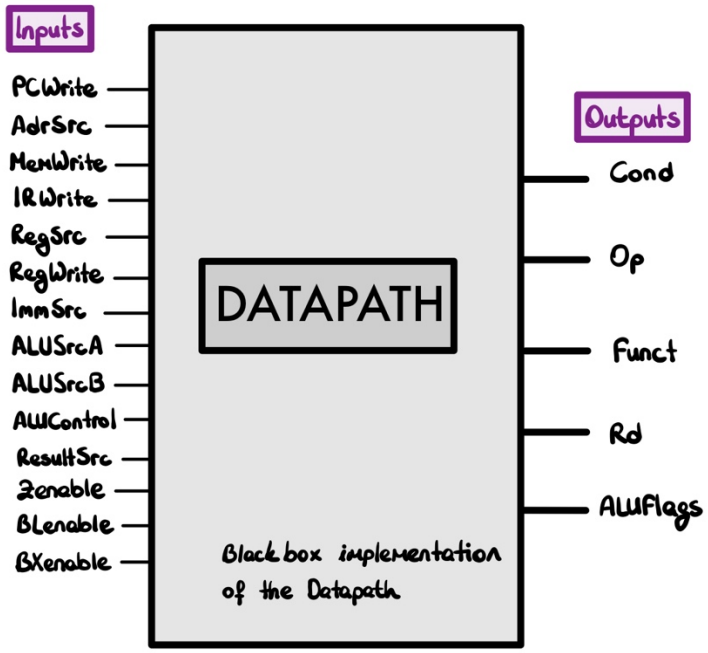


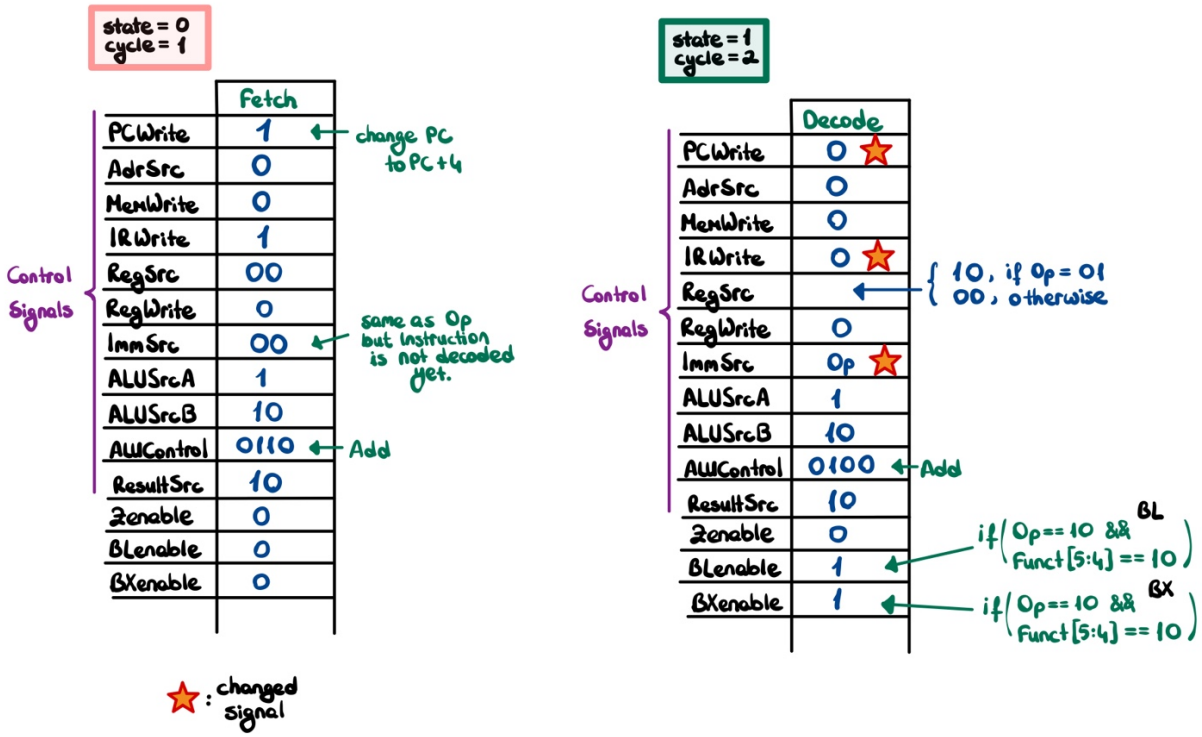
Figure 2: Multi cycle processor from the lecture notes

Black-box implementation of the Datapath



Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

Control Signals



state = 2
cycle = 3

Control Signals

	op = 00 Data P.	op = 01 Memory	op = 10 Branch
PCWrite	0	0	1 ★
AdrSrc	0	0	0
MemWrite	0	0	0
IRWrite	0	0	0
RegSrc	00	10 ○	10
RegWrite	0	0	0
ImmSrc	00	01	10
ALUSrcA	0 ★	0 ★	0 ★
ALUSrcB	00 ★	01 ★	01 ★
ALUControl	Funct[4:1] ★	0100	1101 ★
ResultSrc	00 ★	00 ★	10
Zenable	1 ★	1 ★	1 ★
BLenable	0	0	1 if BL
BXenable	0	0	1 if BX

if CondEx = 1

○ For both LDR and STR
Rn must be chosen.
For STR, Rd must be
connected to RA2 (10)
No need for Rm.

state = 3
cycle = 4

Control Signals

	op = 00 Data P.	op = 0 Memory
PCWrite	0	0
AdrSrc	0	1
MemWrite	0	~Funct[0] ★ ●
IRWrite	0	0
RegSrc	00	10
RegWrite	1 ★	0
ImmSrc	00	01
ALUSrcA	0	0
ALUSrcB	00	01
ALUControl	Funct[4:1]	0100
ResultSrc	00	00
Zenable	1 ★	1 ★
BLenable	0	0
BXenable	0	0

if Funct[4:1] = 1010 (CMP)
RegWrite = 0
ALUControl = 0010 (Subtraction)

● MemWrite is 1 for STR and 0 for LDR.
L bit (Funct[0]) is 0 for STR, 1 for LDR.

state = 4
cycle = 5

Control Signals

	LDR
PCWrite	0
AdrSrc	1
MemWrite	0 ★
IRWrite	0
RegSrc	00
RegWrite	1 ★
ImmSrc	01
ALUSrcA	0
ALUSrcB	01
ALUControl	0100
ResultSrc	01
Zenable	0 ★
BLenable	0
BXenable	0

Write to register

ADD Operation (Data Processing Instruction)

```
[Cycle  1 ]

 Wires:

PC: 00000000000000000000000000001000
State: 000
INSTR: 11100100000100000011000001101000
-----
RA1: 0000
RA2: 1000
-----
RD1: 00000000000000000000000000000000
RD2: 00000000000000000000000000000000
-----
SrcA: 00000000000000000000000000001000
SrcB: 00000000000000000000000000000100
ExtImm: 00000000000000000000000011101000
-----
ALU OUT: 00000000000000000000000001101000
-----
RESULT: 00000000000000000000000000001100

Control Signals:

PCWrite: 1
AdrSrc: 0
MemWrite: 0
IRWrite: 1
RegWrite: 0
ImmSrc: 00
RegSrc: 00
ALUSrcA: 1
ALUSrcB: 10
ALUControl: 0100
ResultSrc: 10
Flag Z: 0
```

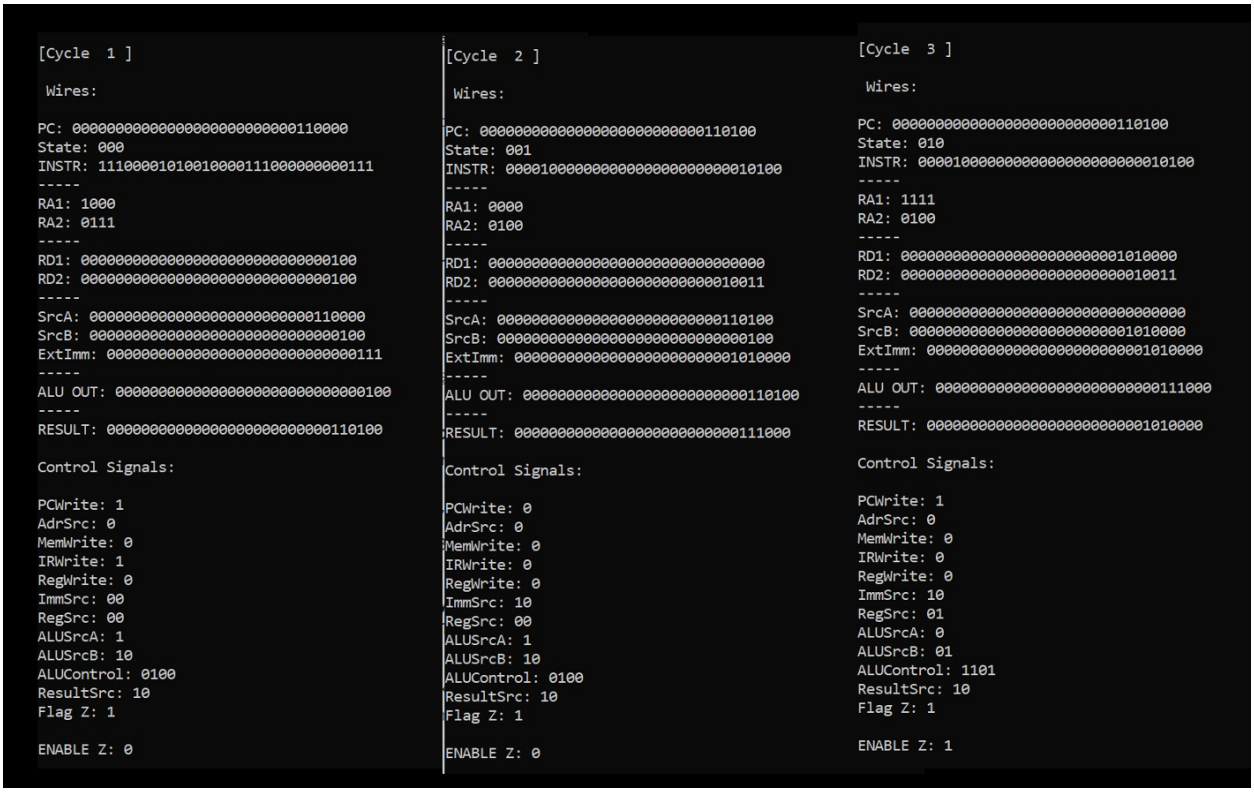
```
[Cycle  2 ]

 Wires:

PC: 00000000000000000000000000001100
State: 001
INSTR: 11100000100000100100000000000011
-----
RA1: 0010
RA2: 0011
-----
RD1: 00000000000000000000000000001001
RD2: 00000000000000000000000000001010
-----
SrcA: 00000000000000000000000000001100
SrcB: 00000000000000000000000000000100
ExtImm: 00000000000000000000000000000011
-----
ALU OUT: 00000000000000000000000000001100
-----
RESULT: 00000000000000000000000000010000

Control Signals:

PCWrite: 0
AdrSrc: 0
MemWrite: 0
IRWrite: 0
RegWrite: 0
ImmSrc: 00
RegSrc: 00
ALUSrcA: 1
ALUSrcB: 10
ALUControl: 0100
ResultSrc: 10
Flag Z: 0

ENABLE Z: 0
```

```
[Cycle  3 ]

 Wires:

PC: 00000000000000000000000000001100
State: 010
INSTR: 11100000100000100100000000000011
-----
RA1: 0010
RA2: 0011
-----
RD1: 00000000000000000000000000001001
RD2: 00000000000000000000000000001010
-----
SrcA: 00000000000000000000000000001001
SrcB: 00000000000000000000000000001010
ExtImm: 00000000000000000000000000000011
-----
ALU OUT: 00000000000000000000000000010000
-----
RESULT: 00000000000000000000000000010000

Control Signals:

PCWrite: 0
AdrSrc: 0
MemWrite: 0
IRWrite: 0
RegWrite: 0
ImmSrc: 00
RegSrc: 00
ALUSrcA: 0
ALUSrcB: 00
ALUControl: 0100
ResultSrc: 00
Flag Z: 0

ENABLE Z: 1
```

```
[Cycle  4 ]

 Wires:

PC: 00000000000000000000000000001100
State: 011
INSTR: 11100000100000100100000000000011
-----
RA1: 0010
RA2: 0011
-----
RD1: 00000000000000000000000000001001
RD2: 00000000000000000000000000001010
-----
SrcA: 00000000000000000000000000001001
SrcB: 00000000000000000000000000001010
ExtImm: 00000000000000000000000000000011
-----
ALU OUT: 00000000000000000000000000010011
-----
RESULT: 00000000000000000000000000010011

Control Signals:

PCWrite: 0
AdrSrc: 0
MemWrite: 0
IRWrite: 0
RegWrite: 1
ImmSrc: 00
RegSrc: 00
ALUSrcA: 0
ALUSrcB: 00
ALUControl: 0100
ResultSrc: 00
Flag Z: 0

ENABLE Z: 1
```

```
[Cycle  5 ]

 Wires:

PC: 00000000000000000000000000001100
State: 100
INSTR: 11100000100000100100000000000011
-----
RA1: 0010
RA2: 0011
-----
RD1: 00000000000000000000000000001001
RD2: 00000000000000000000000000001010
-----
SrcA: 00000000000000000000000000001001
SrcB: 00000000000000000000000000001010
ExtImm: 00000000000000000000000000000011
-----
ALU OUT: 00000000000000000000000000010011
-----
RESULT: 00000000000000000000000000010011

Control Signals:

PCWrite: 0
AdrSrc: 0
MemWrite: 0
IRWrite: 0
RegWrite: 0
ImmSrc: 00
RegSrc: 00
ALUSrcA: 0
ALUSrcB: 00
ALUControl: 0000
ResultSrc: 00
Flag Z: 0

ENABLE Z: 0

### End of instruction ###

Registers:
R2 <-- 9
R3 <-- 10
R4 <-- R2 + R3 = 19
```

BEQ (Branch Instruction)



```
[Cycle  1 ]

 Wires:

PC: 00000000000000000000000000110000
State: 000
INSTR: 11100001010010000111000000000111
-----
RA1: 1000
RA2: 0111
-----
RD1: 00000000000000000000000000000100
RD2: 00000000000000000000000000000100
-----
SrcA: 00000000000000000000000000110000
SrcB: 00000000000000000000000000000100
ExtImm: 00000000000000000000000000000111
-----
ALU OUT: 00000000000000000000000000000100
-----
RESULT: 00000000000000000000000000110100

Control Signals:

PCWrite: 1
AdrSrc: 0
MemWrite: 0
IRWrite: 1
RegWrite: 0
ImmSrc: 00
RegSrc: 00
ALUSrcA: 1
ALUSrcB: 10
ALUControl: 0100
ResultSrc: 10
Flag Z: 1

ENABLE Z: 0
```

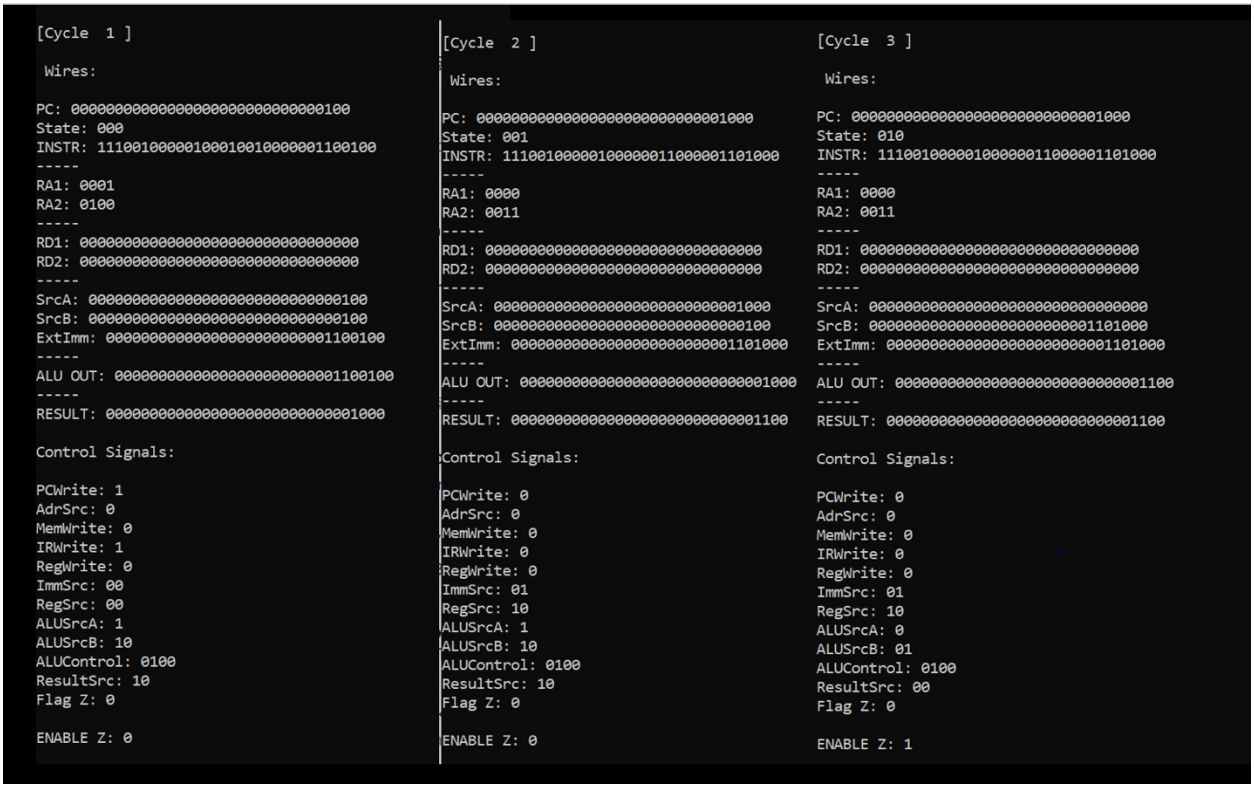
```
[Cycle  2 ]

 Wires:

PC: 00000000000000000000000000110100
State: 001
INSTR: 00001000000000000000000000010100
-----
RA1: 0000
RA2: 0100
-----
RD1: 00000000000000000000000000000000
RD2: 00000000000000000000000000010011
-----
SrcA: 00000000000000000000000000110100
SrcB: 00000000000000000000000000000100
ExtImm: 00000000000000000000000001010000
-----
ALU OUT: 00000000000000000000000000110100
-----
RESULT: 00000000000000000000000000111000

Control Signals:

PCWrite: 0
AdrSrc: 0
MemWrite: 0
IRWrite: 0
RegWrite: 0
ImmSrc: 10
RegSrc: 00
ALUSrcA: 1
ALUSrcB: 10
ALUControl: 0100
ResultSrc: 10
Flag Z: 1

ENABLE Z: 0
```

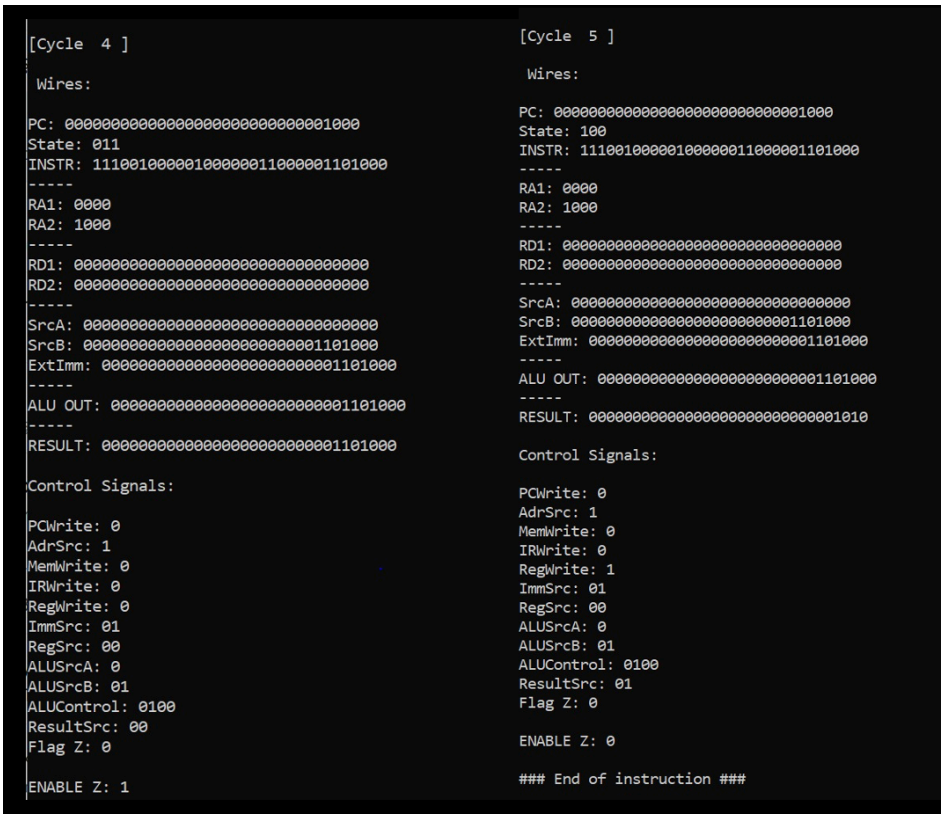
```
[Cycle  3 ]

 Wires:

PC: 00000000000000000000000000110100
State: 010
INSTR: 00001000000000000000000000010100
-----
RA1: 1111
RA2: 0100
-----
RD1: 00000000000000000000000001010000
RD2: 00000000000000000000000000010011
-----
SrcA: 00000000000000000000000000000000
SrcB: 00000000000000000000000001010000
ExtImm: 00000000000000000000000001010000
-----
ALU OUT: 00000000000000000000000000111000
-----
RESULT: 00000000000000000000000001010000

Control Signals:

PCWrite: 1
AdrSrc: 0
MemWrite: 0
IRWrite: 0
RegWrite: 0
ImmSrc: 10
RegSrc: 01
ALUSrcA: 0
ALUSrcB: 01
ALUControl: 1101
ResultSrc: 10
Flag Z: 1

ENABLE Z: 1
```

```
[Cycle  4 ]

 Wires:

PC: 00000000000000000000000001010000
State: 011
INSTR: 00001000000000000000000000010100
-----
RA1: 1111
RA2: 0100
-----
RD1: 00000000000000000000000001010000
RD2: 00000000000000000000000000010011
-----
SrcA: 00000000000000000000000001010000
SrcB: 00000000000000000000000001010000
ExtImm: 00000000000000000000000001010000
-----
ALU OUT: 00000000000000000000000001010000
-----
RESULT: 00000000000000000000000001010000

Control Signals:

PCWrite: 0
AdrSrc: 0
MemWrite: 0
IRWrite: 0
RegWrite: 0
ImmSrc: 10
RegSrc: 01
ALUSrcA: 0
ALUSrcB: 01
ALUControl: 1101
ResultSrc: 10
Flag Z: 0

ENABLE Z: 1
```

```
[Cycle  5 ]

 Wires:

PC: 00000000000000000000000001010000
State: 100
INSTR: 00001000000000000000000000010100
-----
RA1: 0000
RA2: 0100
-----
RD1: 00000000000000000000000000000000
RD2: 00000000000000000000000000010011
-----
SrcA: 00000000000000000000000001010000
SrcB: 00000000000000000000000000010011
ExtImm: 00000000000000000000000000010100
-----
ALU OUT: 00000000000000000000000001010000
-----
RESULT: 00000000000000000000000001010000

Control Signals:

PCWrite: 0
AdrSrc: 0
MemWrite: 0
IRWrite: 0
RegWrite: 0
ImmSrc: 00
RegSrc: 00
ALUSrcA: 0
ALUSrcB: 00
ALUControl: 0000
ResultSrc: 00
Flag Z: 0

ENABLE Z: 0

### End of instruction ###
```

LDR operation (Memory Instruction)





PC	Instruction	Hex	Comment
[0]	**LDR R2, [R1, #100];**	E4112064	R2 <-- 00000009
[4]	**B 40;**	E800000A	Go to the 2's Complement Subroutine
-			
~ 2's Complement Subroutine ~			
[40]	**SUB R2, R1, R2;**	E0412002	R2 <-- FFFFFFF7
[44]	**B 8;**	E8000002	
-			
[8]	**LDR R3, [R1, #104];**	E4113068	R3 <-- 3
[12]	**LDR R4, [R1, #108];**	E411406C	R4 <-- 4
[16]	**LDR R1, [R0, #112];**	E4101070	R1 <-- 1
[20]	**LDR R5, [R0, #116];**	E4105074	R5 <-- 120
[24]	**B 64;**	E8000010	Go to the Sum of an Array Subroutine
-			
~ Sum of Array Subroutine ~			
[64]	**LDR R6, [R5];**	E4156000	R6 <-- 1A (first array element) (2B second) (3C third)
[68]	**ADD R10, R10, R6;**	E08AA006	R10 <-- R6 + R10 (sum)
[72]	**SUB R3, R3, R1;**	E0433001	R3 <-- R3 - R1 (decrement counter by 1)
[76]	**CMP R3, R3, R0;**	E1433000	R3 <-- R3 - R0 (Check Flag)
[80]	**BEQ 28;**	08000007	if Z = 1, return
[84]	**ADD R5, R5, R4;**	E0855004	R5 <-- R5 + R4 (increment R5 by 4 to get the next element)
[88]	**B 64;**	E8000010	Start the loop again
-			
[28]	**LDR R9, [R3, #120];**	E4139078	R9 <-- 1A
[32]	**B 132;**	E8000021	Go to the Even Parity Check Subroutine
-			
~ Even Parity Check Subroutine ~			
[132]	**CMP R9, R9, R3;**	E1499003	R9 <-- R9 - R3 (Check Flag)
[136]	**BEQ 160;**	08000028	if Z = 1, go to the final stage
[140]	**AND R0, R9, R1;**	E0090001	R0 <-- R9 & R1 (Get the LSB)
[144]	**ADD R8, R8, R0;**	E0888000	R8 <-- R8 + R0 (Compute the number of ones)
[148]	**MOV R9, R9, #1;**	E1A090A9	R9 <-- R9>>1 (Get the next bit)
[152]	**B 132;**	E8000021	Start the loop again
-			
[160]	**AND R0, R8, R1;**	E0080001	R0 <-- R8 & R1 (if 1, number of ones is odd)
[164]	**B 0;**	E8000000	Return

~ Data written in memory ~

[100]	9	[104]	3	[108]	4	[112]	1
[116]	78	[120]	1A	[124]	2B	[128]	3C

In this experiment, we are implementing a multi-cycle processor design. In the previous experiment, where-single cycle processor is used, each instruction was done in one cycle. In multi-cycle processor design, each instruction is finalized in multiple cycles. However, at one cycle, operations take less longer comparing to the single-cycle processor. Datapath is partially used in each cycle.

In my design, number of cycles are set to be 5 for each instruction. Longest instruction set the number of cycles per instruction and LDR (longest instruction) takes 5 cycles.

In the first cycle of every instruction, **FETCH**, PC is fed to the Instr/Data Memory and PC + 4 is written in PC register. In the second cycle of every instruction, **DECODE**, instruction is read from the PC address.

After DECODE, the instruction is known, and the control signals are set accordingly. Control signals can be observed for each type of instruction above.

Setting the control signals **(CONTROLLER.v)** for each instruction regarding the Datapath **(DATAPATH.v)**, all the instructions are tested through COCOTB. Test procedure for all ISA instructions can be observed in **ISA_TEST_COCO** file. In this report, implementation of the ADD, BEQ, and LDR instructions are shown with the control signals and wire outputs at each cycle.

After validating the functionality of each instruction, three subroutines are implemented. These subroutines are 2's Complement, Sum of Array and Even Parity Check. The algorithm for these operations can be observed above, where all the necessary instructions are stated for each subroutine with their values in HEX and additional comments. To monitor the testing procedure, **SUBROUTINE_TEST** file can be inspected.

COCOTB tests are located in ISA_TEST_COCO and SUBROUTINE_TEST files.