B38DF
Computer Architecture and Embedded Systems

Alexander Belyaev

Heriot-Watt University
School of Engineering & Physical Sciences
Electrical, Electronic and Computer Engineering

E-mail: a.belyaev@hw.ac.uk
Office: EM2.29

Based on the slides provided by Dr. Paul Record and Dr. Pat Chambers

Part 2 Planning

- **Week 7**: Intro to HWU RISC machine
- **Week 8**: Verilog remainder lectures
- **Week 9**: Branching code, conditional code
- **Week 10**: Commercial microcontrollers, Atmel controller. Programming controllers
- **Week 11**: More on programming microcontrollers, microchip controller
- **Week 12**: Revision

Programming exercises + a programming project

1. Simple circuit design with logisim programming exercise (2 marks, week 7)
2. Three verilog programming exercises with Verilog (3 marks, week 8)
3. A programming project on a verilog simulation of the HWU machine (10 marks, weeks 9-12)

A class test (10 marks, week 11)

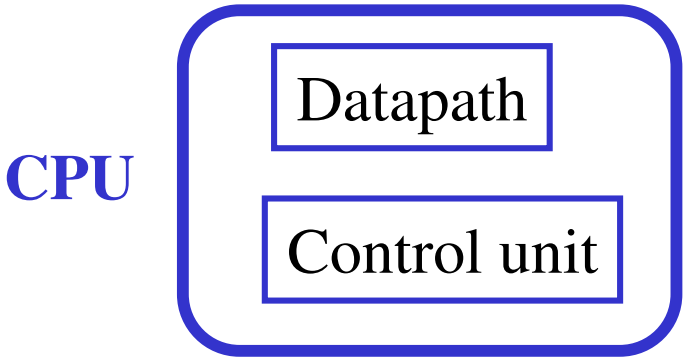
Programmable Machines

A programmable machine and has two general properties

1. **It responds to a set of instruction in a well defined way**
2. **It is able to 'replay' these instructions as a programme.**

The core part of a computer is the central processing unit, CPU

CPUs are just complex finite state machines. The advent of the single chip processor is the result of advances in the design and implementation of large digital circuits.

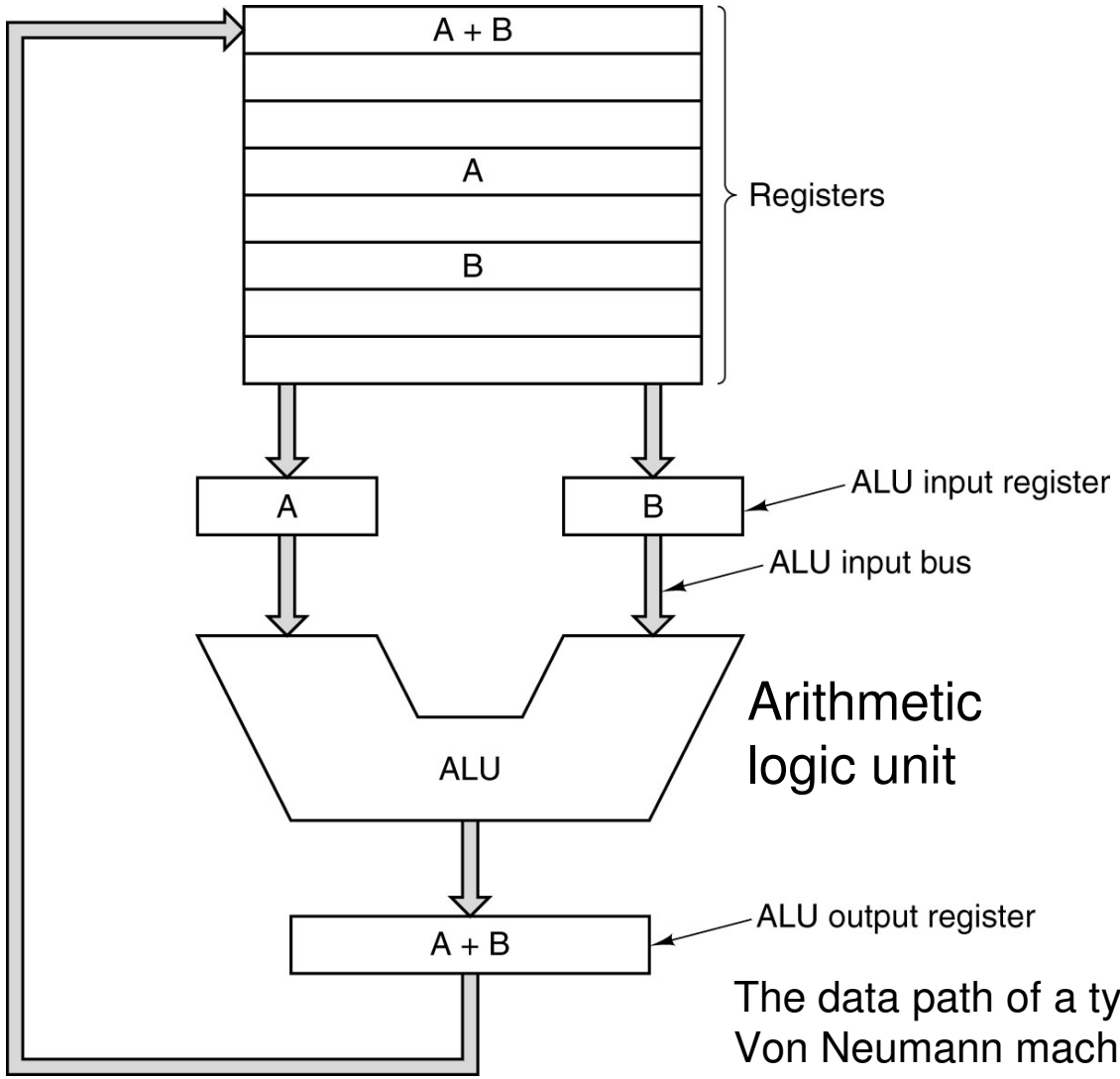


Datapath = Registers + ALU + Connection Buses

Datapath
=
Registers
+
ALU
+
Connection Buses

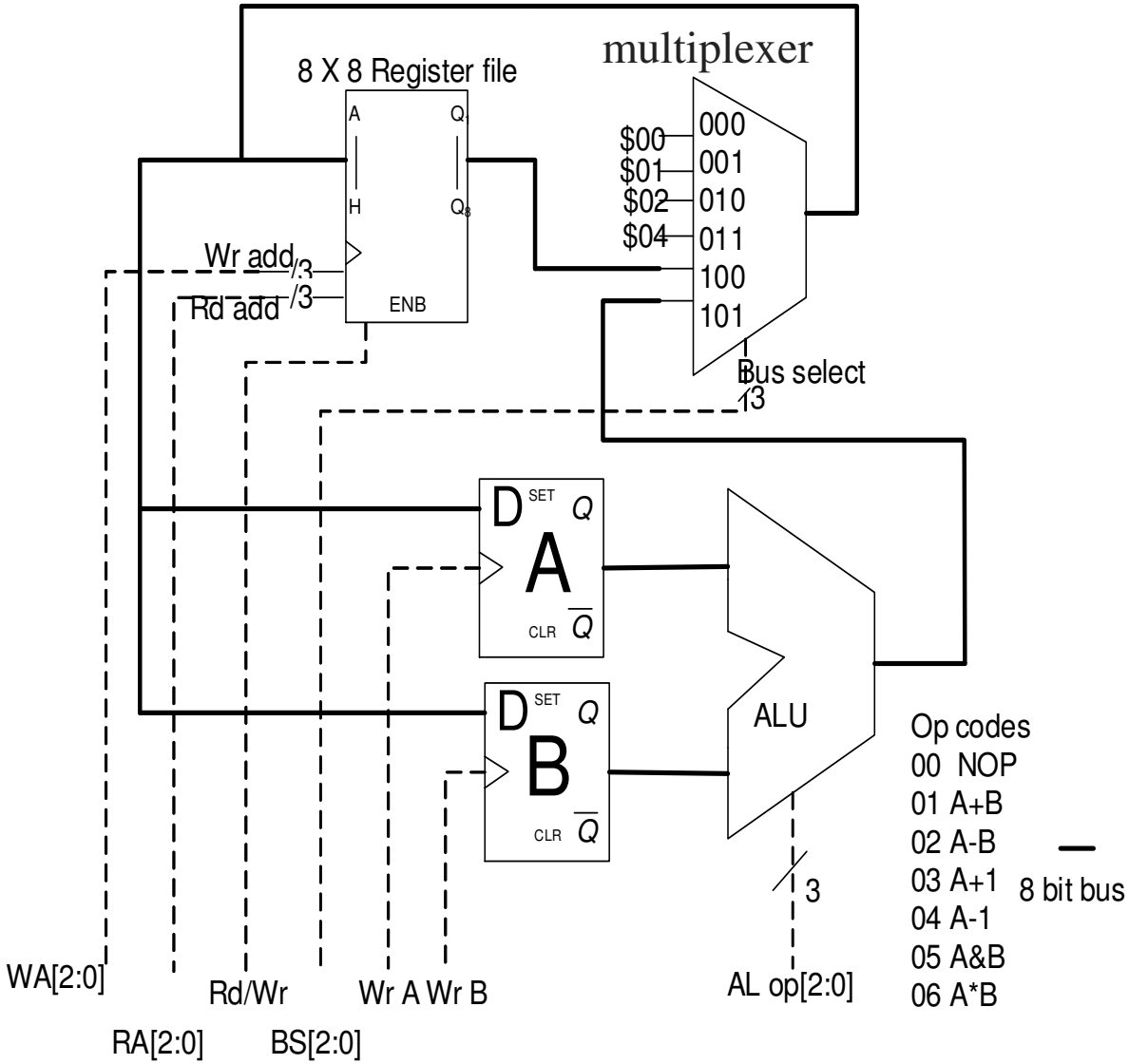
Datapath Cycle:
The process of running two operands through the ALU and storing the result back in the registers

→The **heart** of most CPUs
→The **faster the data path** the faster the machine runs



The data path of a typical Von Neumann machine

A simple datapath studied last semester (within B38DB)



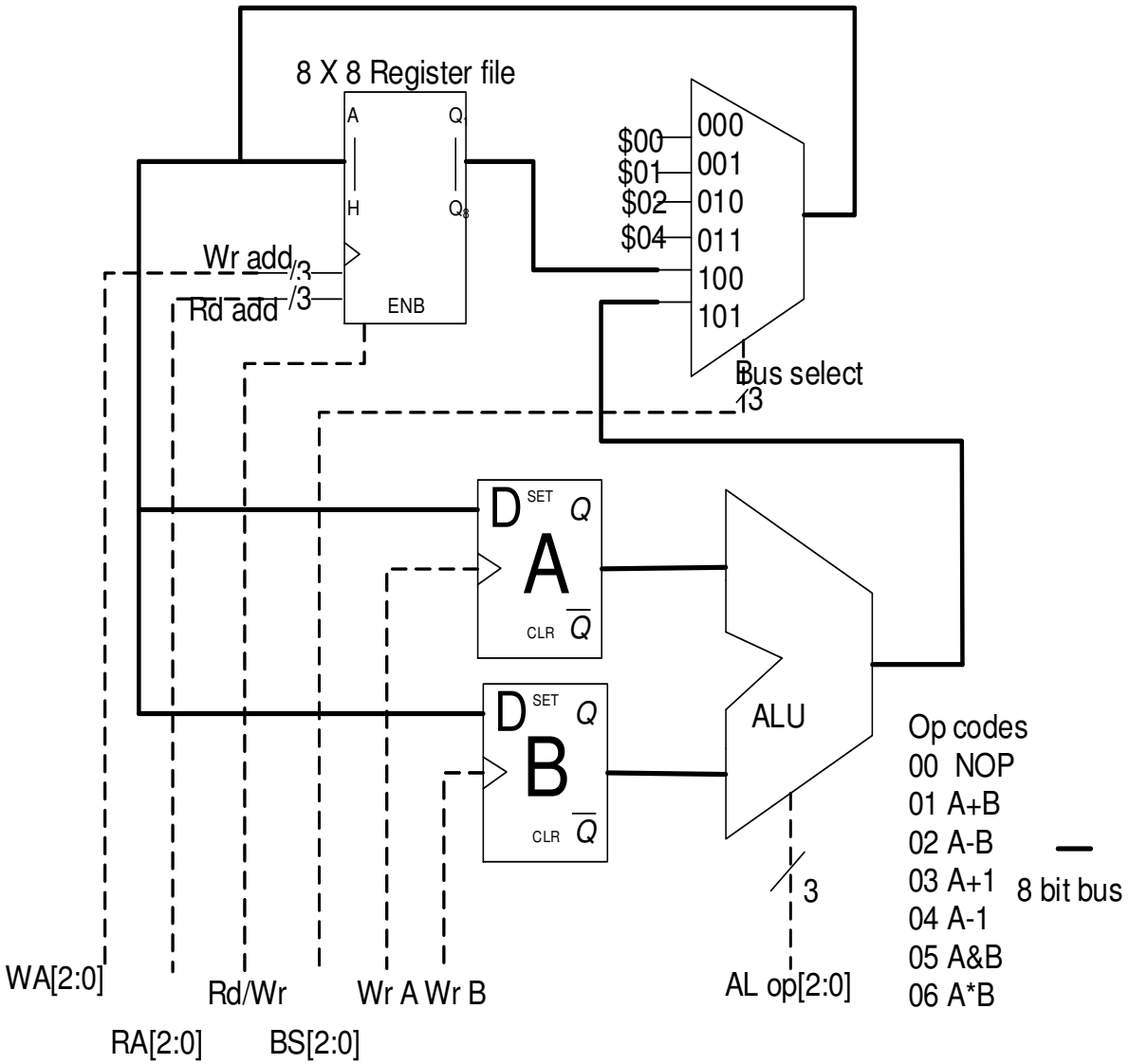
(i) A Register file: a form of memory that has limited addressing and where instructions and operands from main memory are placed. Its control lines are:

WA[2:0] : a 3-bit line specifying an address to which data will be written;

RA[2:0] : a 3-bit line specifying an address from which data will be read;

Rd/Wr : a 1-bit line specifying whether data is being read or written. 0 means write, 1 means read.

A simple datapath studied last semester (within B38DB)



(ii) A set of bus lines (thick black lines): These pass data around the CPU and can be several (in this case 8) bits wide;

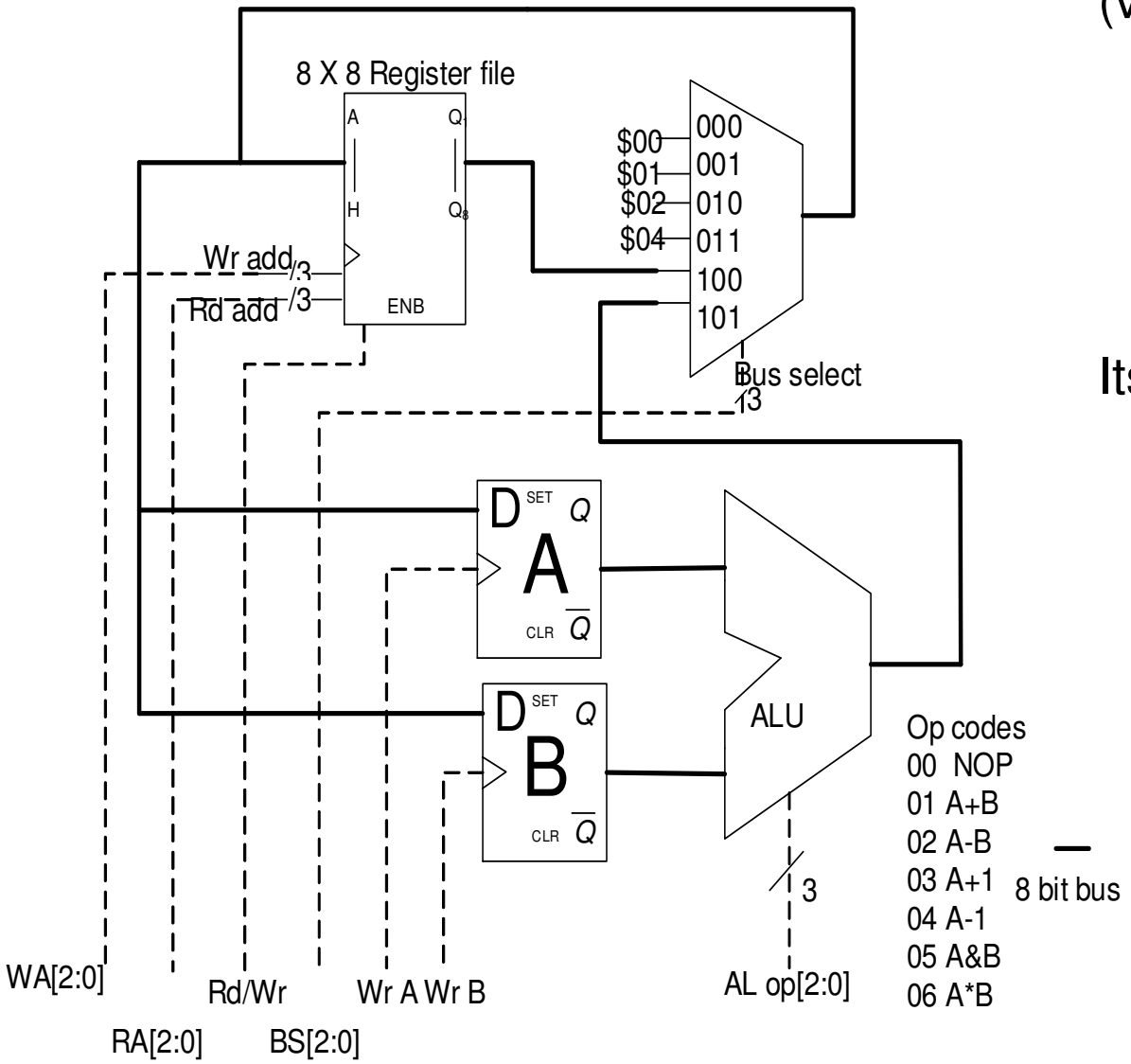
(iii) Bus select: An element that controls the flow of data in the bus lines. Its control line is:

BS[2:0] : A 3 bit number that specifies which bus line to activate (101,100). The bits sets: 000, 001, 010 and 011 are used to output numerical constants: 0, 1, 2, and 4 onto the bus

(iv) A and B are the buffers to hold operands just before the ALU.

Their control lines: WrA and WrB are 1 bit lines notifying A and B when they are to be written.

A simple datapath studied last semester (within B38DB)



(v) The arithmetic logic unit (ALU) performs various operations on A and/or B based on the opcodes it receives from the instruction at hand.

Its control line: ALop[2:0] is a 3 bit line specifying which opcode is to be performed.

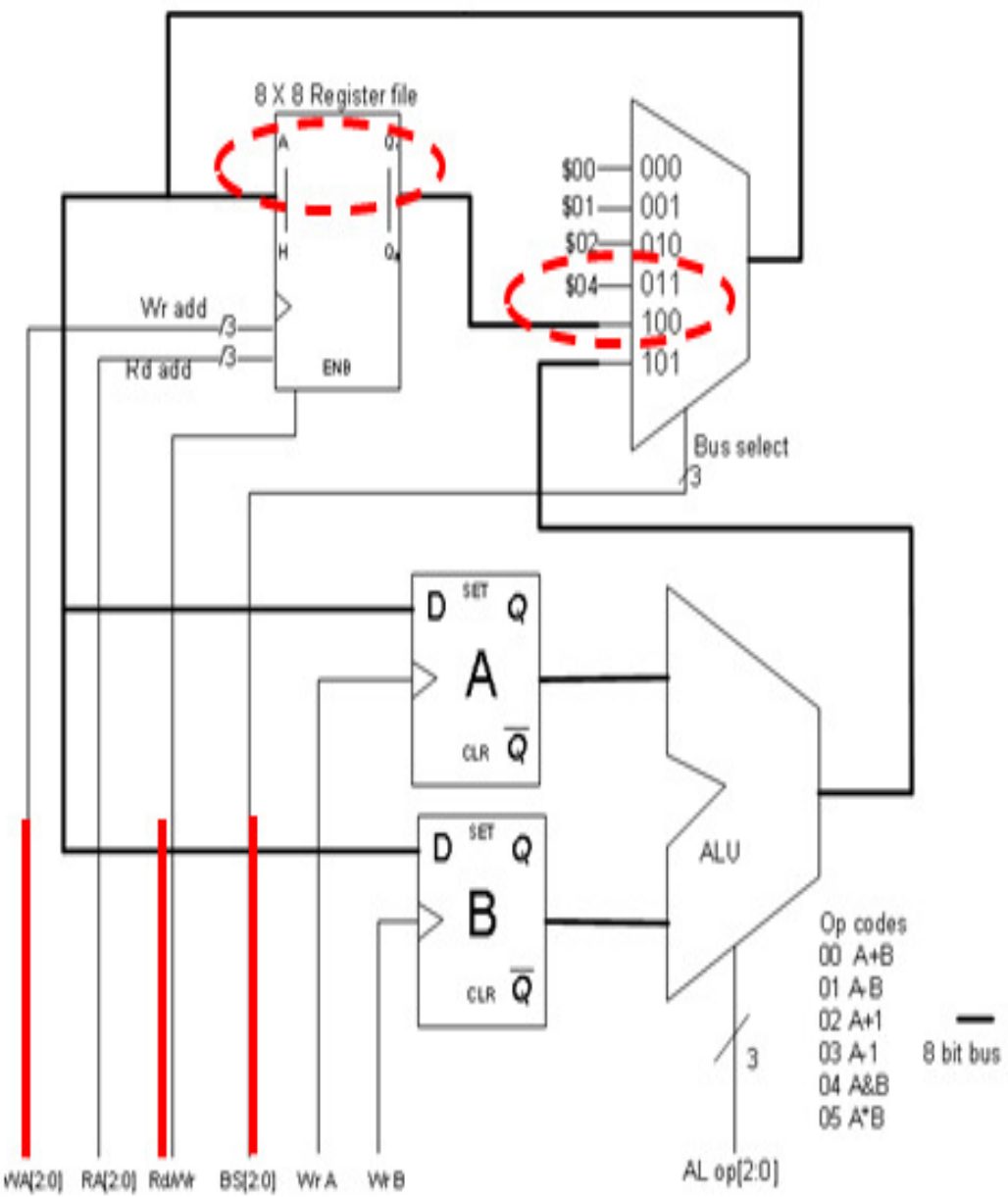
A simple datapath studied last semester (within B38DB)

Instructions

The instructions are assumed to follow a format where an operation is specified and the result overwrites the first variable stated.

Example 1: R1 = 4

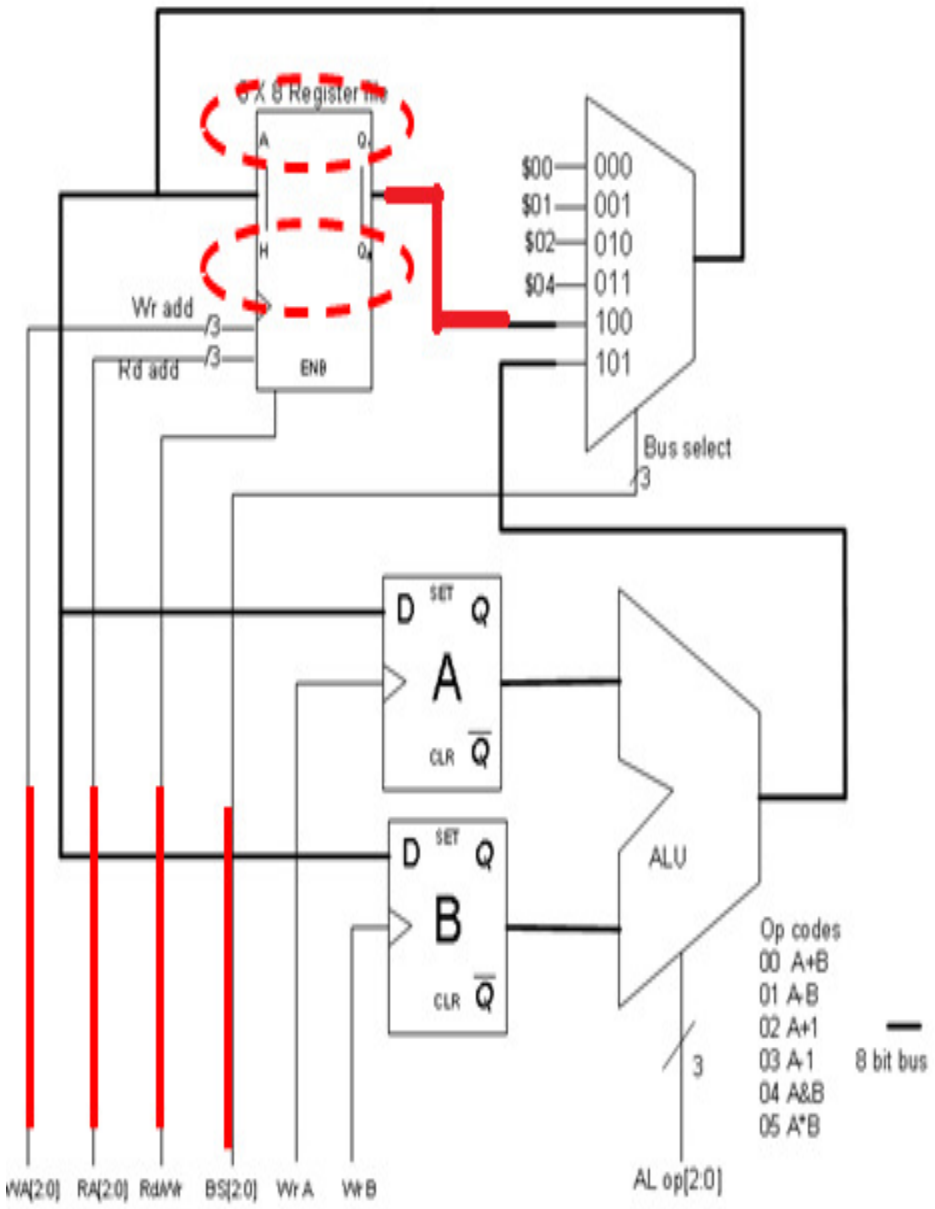
Step	Operation	Comment
1	Set BS = 011	On bus select (BS), a constant, 4, is outputted.
2	Set WA = 001	R1 corresponds to a write address (WA) of 1.
3	Set Rd/Wr = 0	To write the 4 to R1.
4	Set Rd/Wr = 1	Stop the writing process.



A simple datapath studied last semester (within B38DB)

Example 2 : R2 = R3

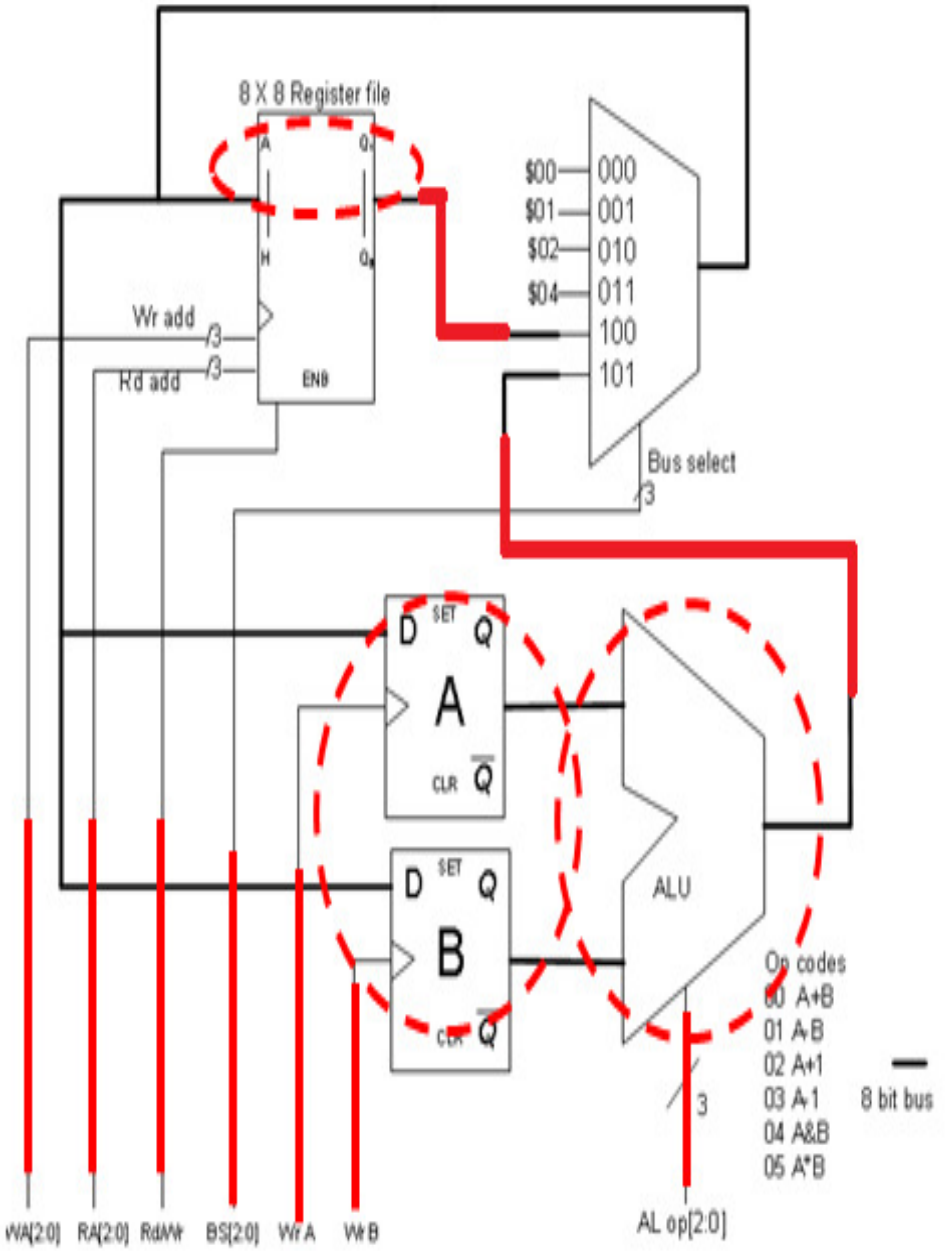
Step	Operation	Comment
1	Set BS = 100	On BS, connect the register file (RF) to the bus line.
2	Set RA = 011	Having connected BS to RF, further stipulate that BS is connected to R3.
3	Set Rd/Wr = 1	Read R3.
4	Set WA = 010	While reading R3, select the write address as R2 so it can start to be written there.
5	Set Rd/Wr = 0	Now starting (R3) to R2.
6	Set Rd/Wr = 1	Stop the writing process.



A simple datapath studied last semester (within B38DB)

Example 3 : R2 = R3 AND R2

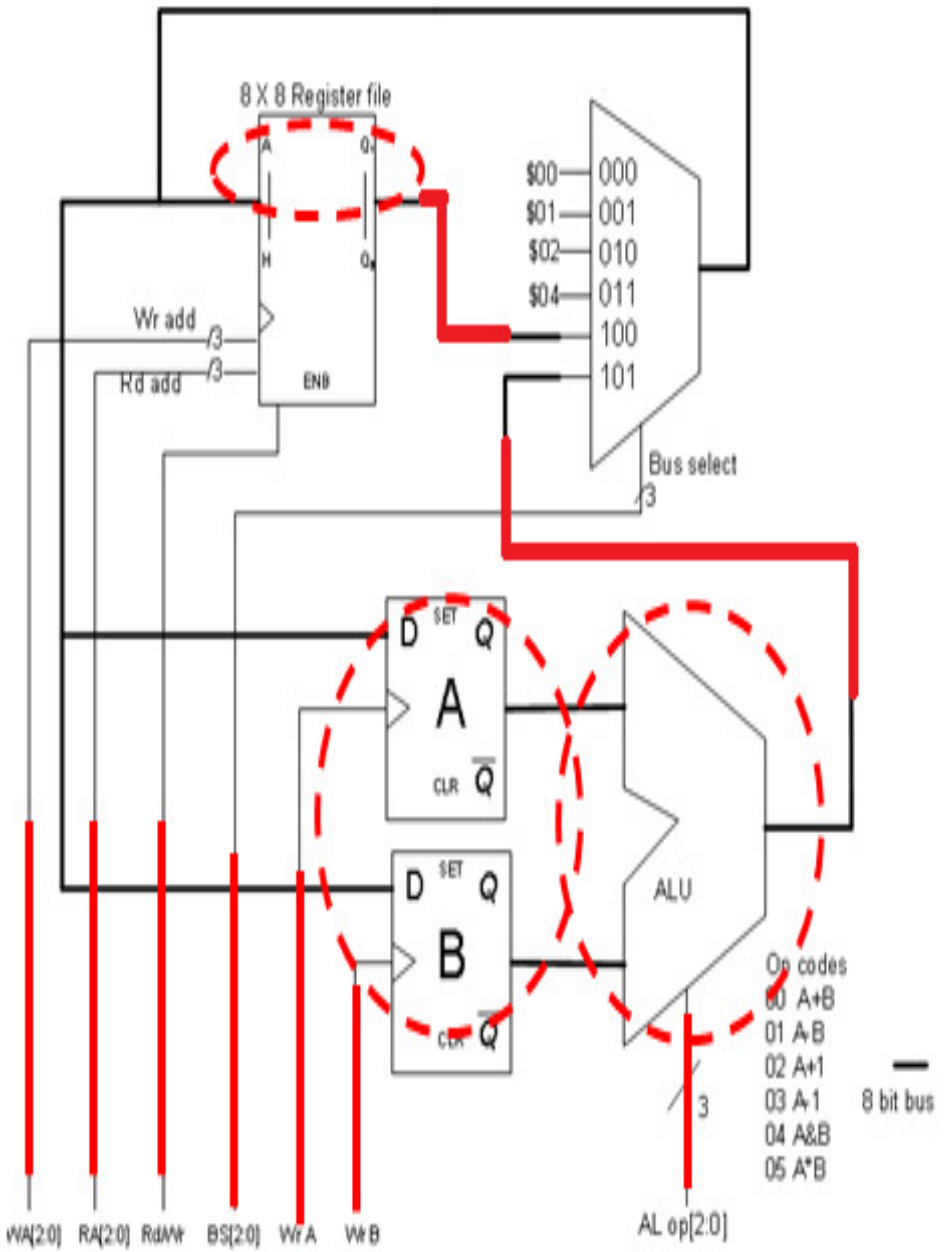
Step	Operation	Comment
1	Set BS = 100	On BS, connect the register file (RF) to the bus line.
2	Set RA = 010	Having connected BS to RF, further stipulate that BS is connected to R2.
3	Set Rd/Wr = 1	Read R2.
4	Set WrA = 1	While reading R2, write it to buffer: A.
5	Set RA = 011	Connect the same bus line from steps 1 and 2 to R3.
6	Set WrB = 1	While reading R3, write it to buffer: B.



A simple datapath studied last semester (within B38DB)

Example 3 : R2 = R3 AND R2

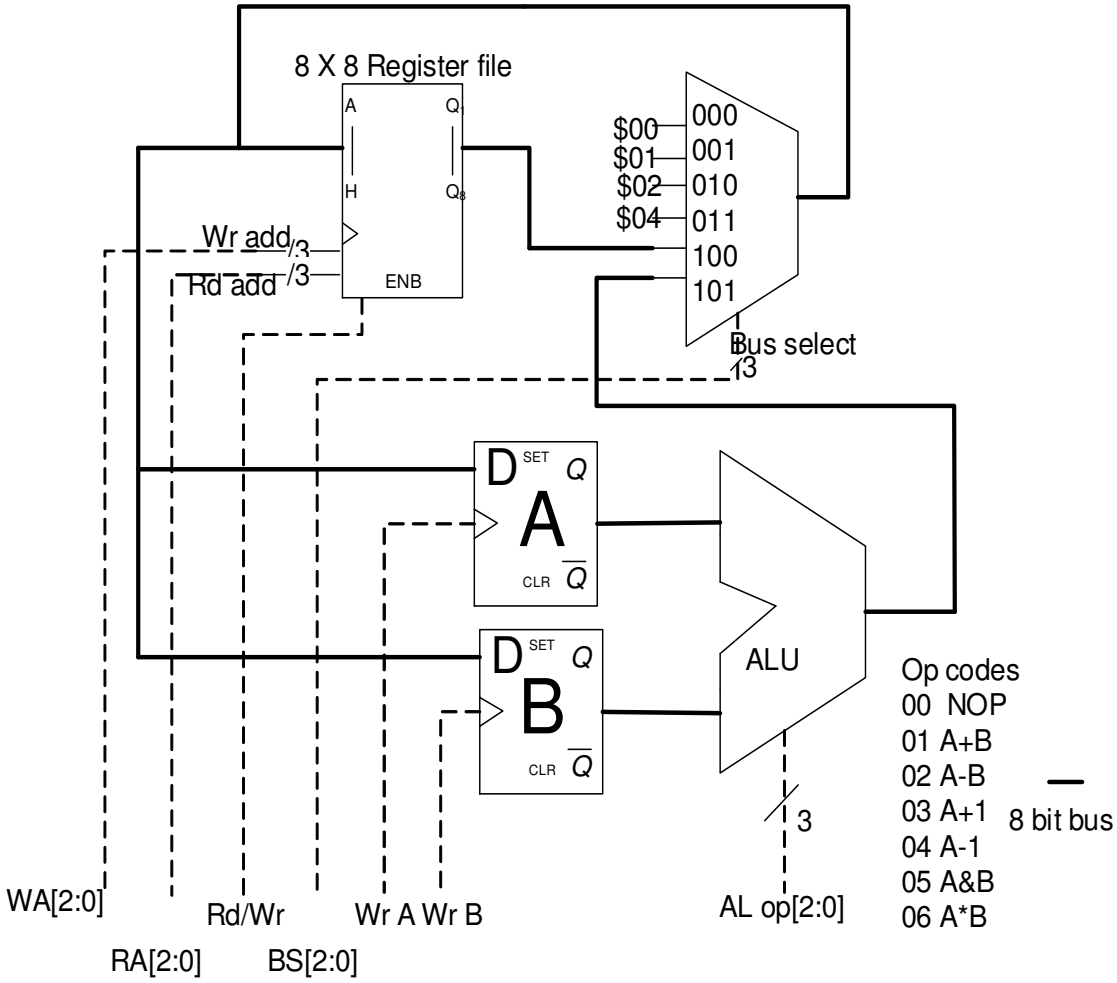
Step	Operation	Comment
7	Set ALU op = 100	It is desired to perform logical AND, the ALU understands this to be 4 or 010.
8	Set BS = 101	On BS, connect the register file (RF) to the ALU.
9	Set WA = 010	Tell the RF that the result from the ALU will be written to R2.
10	Set Rd/Wr = 0	Write to R2
11	Set Rd/Wr = 1	Stop the writing process.



A simple datapath studied last semester (within B38DB)

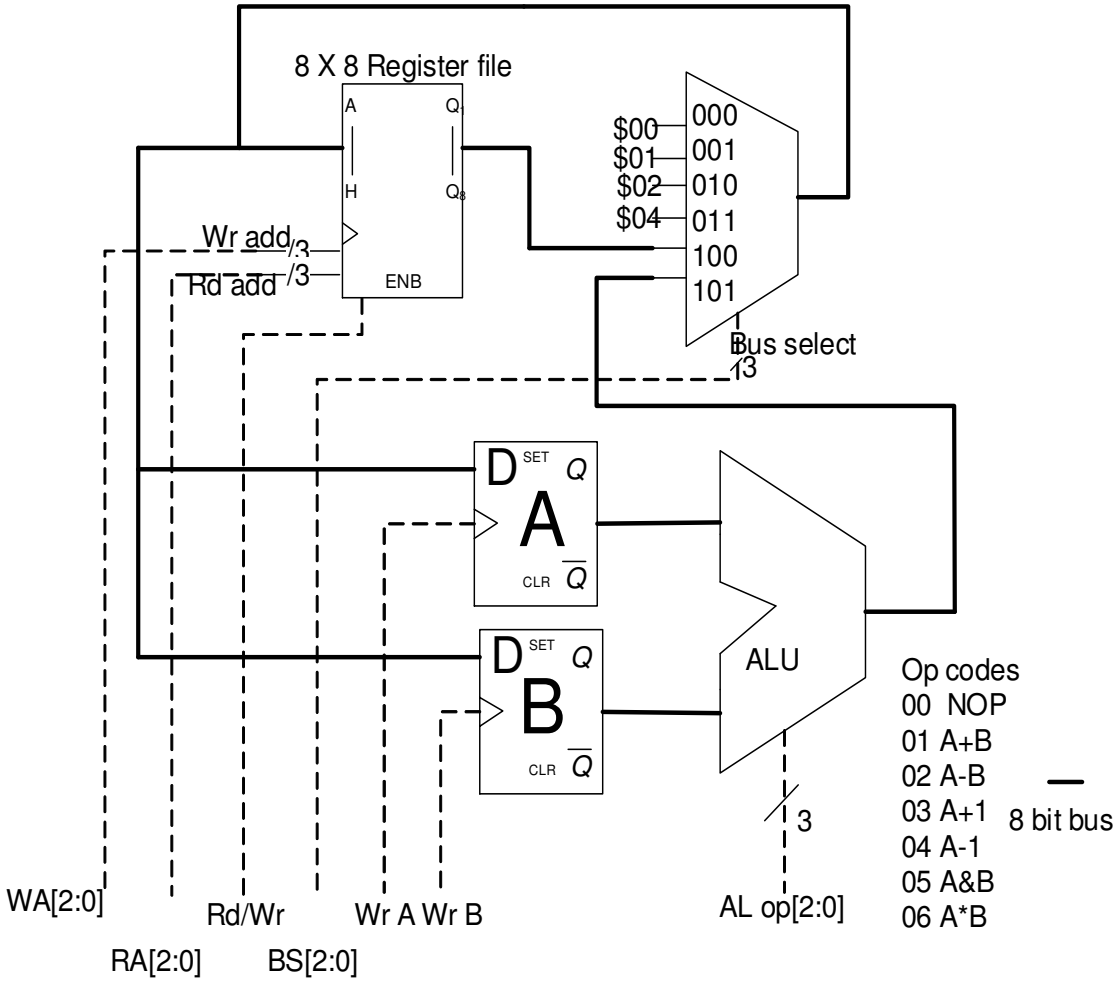
Example 4 : R2 = R3 + 2

Sequence of steps:
1. Set BS = 100 (reg out connected to bus)
2. Set RA = 011 (bus connected to reg out connected to R3)
3. Set Rd/Wr = 1 (read R3)
4. Set WrA = 1 (write R3 contents into A)
5. Set BS = 010
6. Set Wr B = 1 (write constant 2 into B
7. Set AL op = 1
8. Set BS = 101
9. Set WA = 010
10. Set Rd/Wr = 0 (write ALU output to R2
11. Set Rd/Wr = 1



A simple datapath studied last semester (within B38DB)

Example 5 : R4 = R4 + 1



A simple datapath studied last semester (within B38DB)

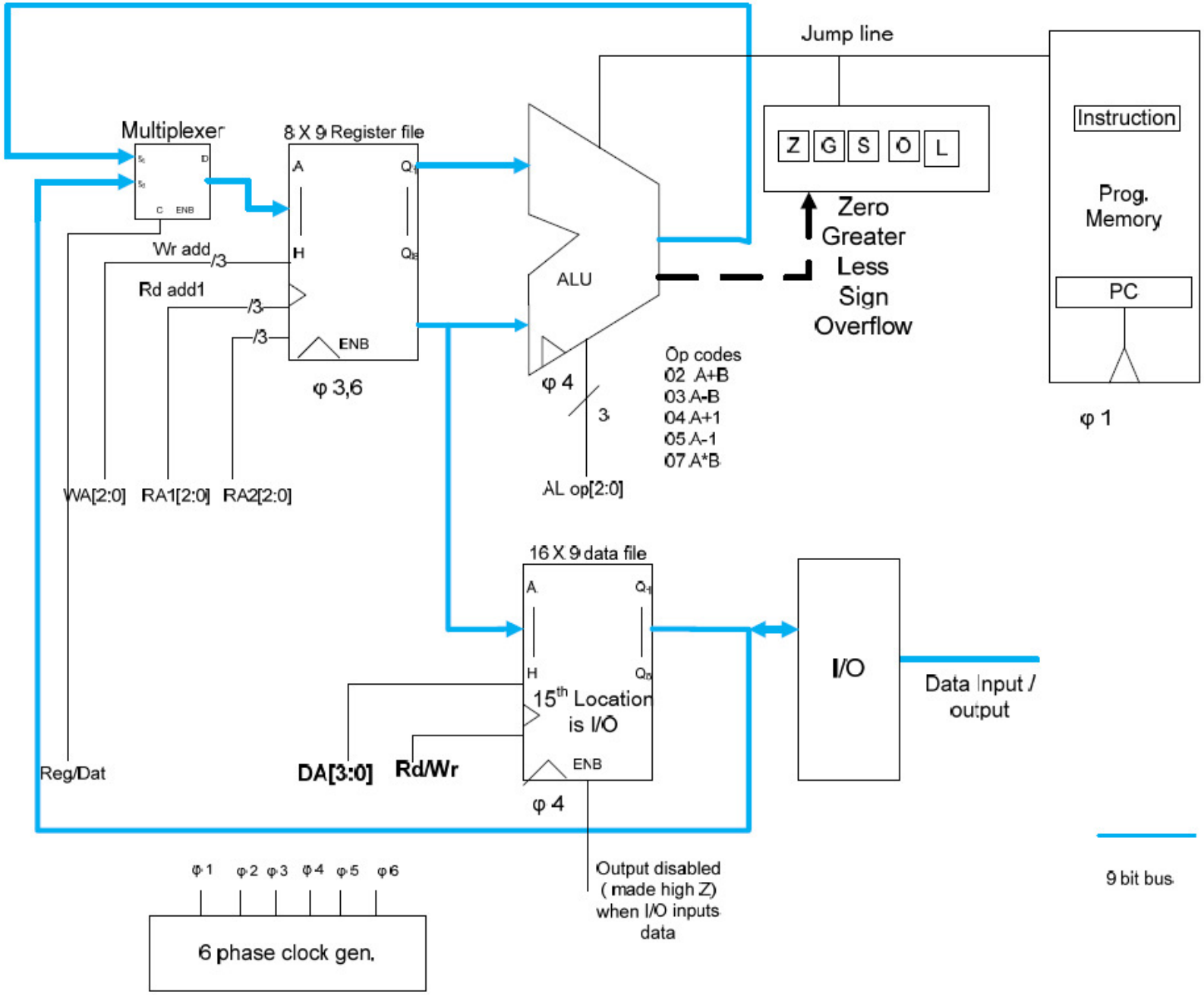
There are a number of drawbacks of the datapath we have considered:

- Limited instructions;
- Excessive use of instructions to simple things;
- Very difficult to do ‘jumps’;
- High-level programming constructs are very difficult to do.

```
for (i=0; i<10; i++)
    sum = sum + A[i]
```

This would require: a counter, and counter increment, access an array, repeat a specific set of instructions over and over until condition met.

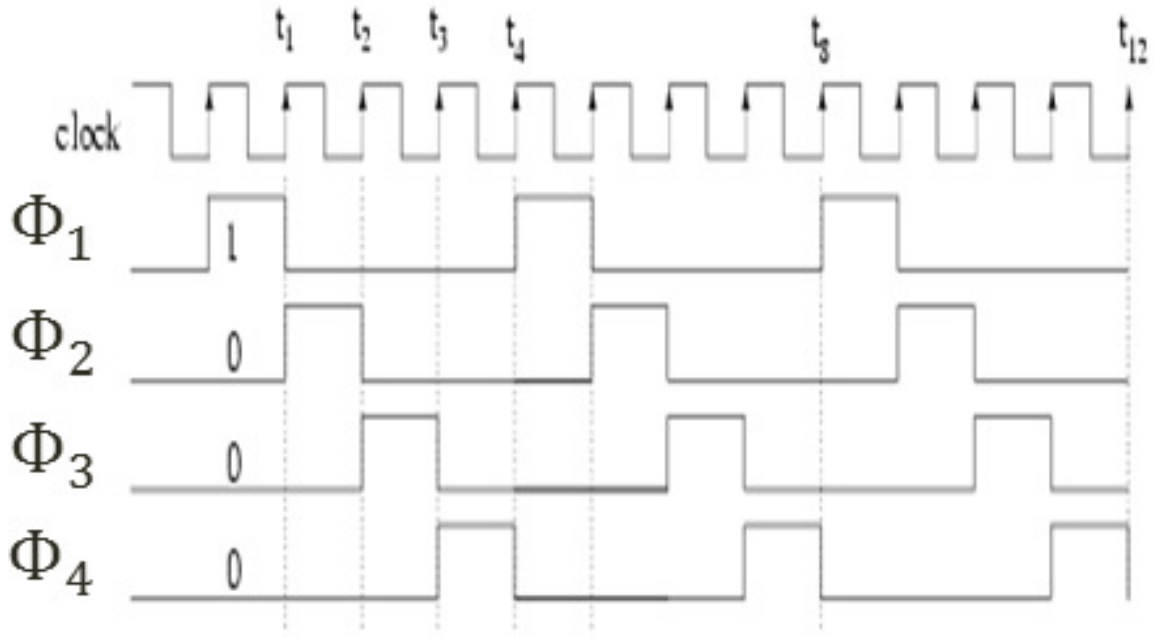
Architecture of the HWU computer datapath with program memory



6 phase clock generator

We need a set of six non-overlapping clock signals. Each of these, Φ_1, Φ_2, Φ_3, Φ_4, Φ_5, Φ_6 is responsible for a certain process. This effectively replaces the bus select system in the simple datapath machine and co-ordinates the device's dataflow.

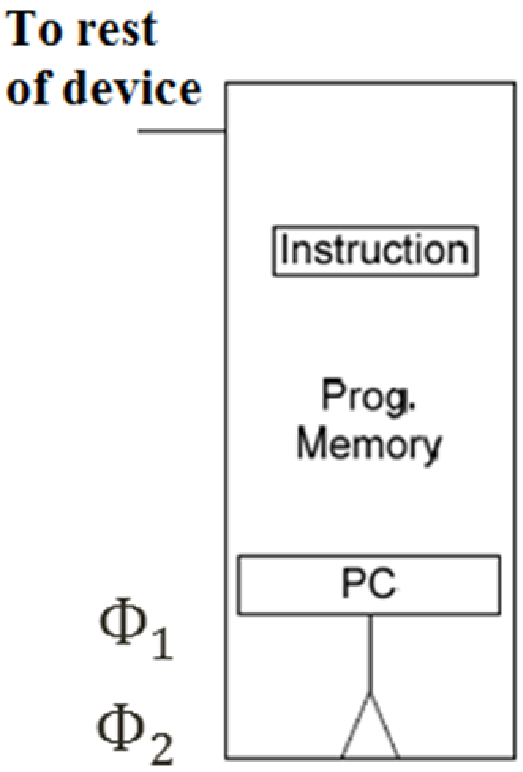
Only four of the six signals are shown here but the pattern is clear. They are each derived from the main clock signal and are equally spaced but non-overlapping.



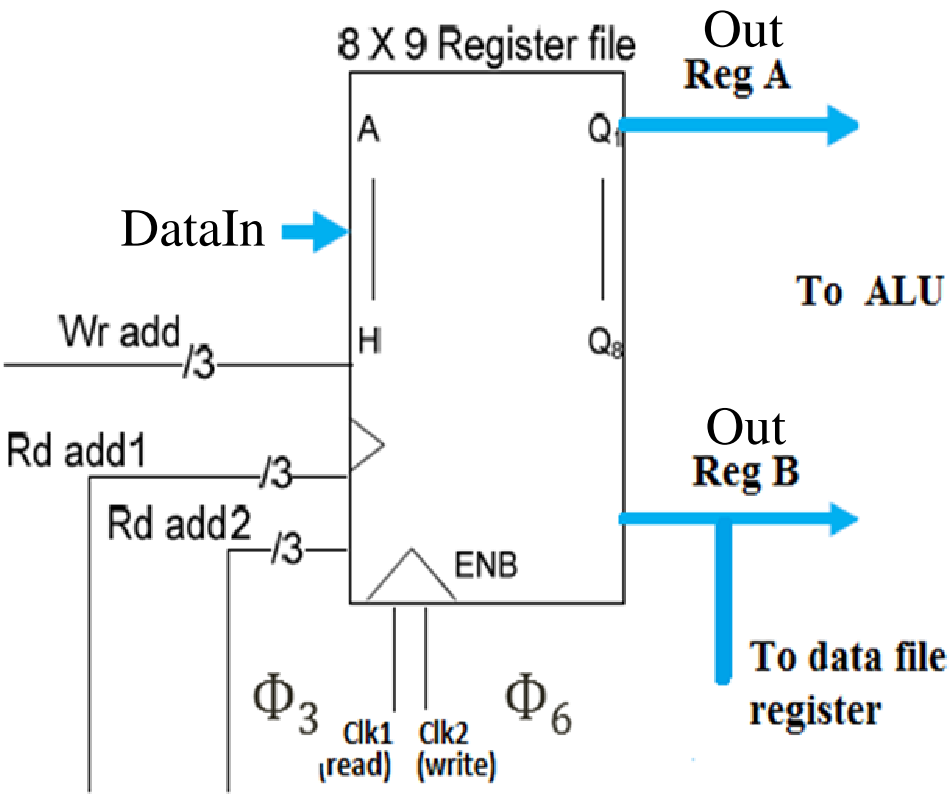
Steps 1 and 2

Use Φ_1 to trigger the program memory (instruction counter).
This effectively loads the instruction.

Use Φ_2 to signal the device to decode the instruction.



Step 3



Accessing 8 registers requires 3 bits

Use Φ_3 to take the decoded instruction and locate and read necessary operands from a register.

This register memory has 8 registers with 9 bits each.

It has one input and two outputs to A and B (similar to other machine).

It accepts two clock signals, one from Φ_3 to read and another from Φ_6 to write. This is part of how this new CPU datapath device is co-ordinated using the clock signals.

The outputs are similar to reg/buffer A and B in the other device and these are passed to the ALU from here.

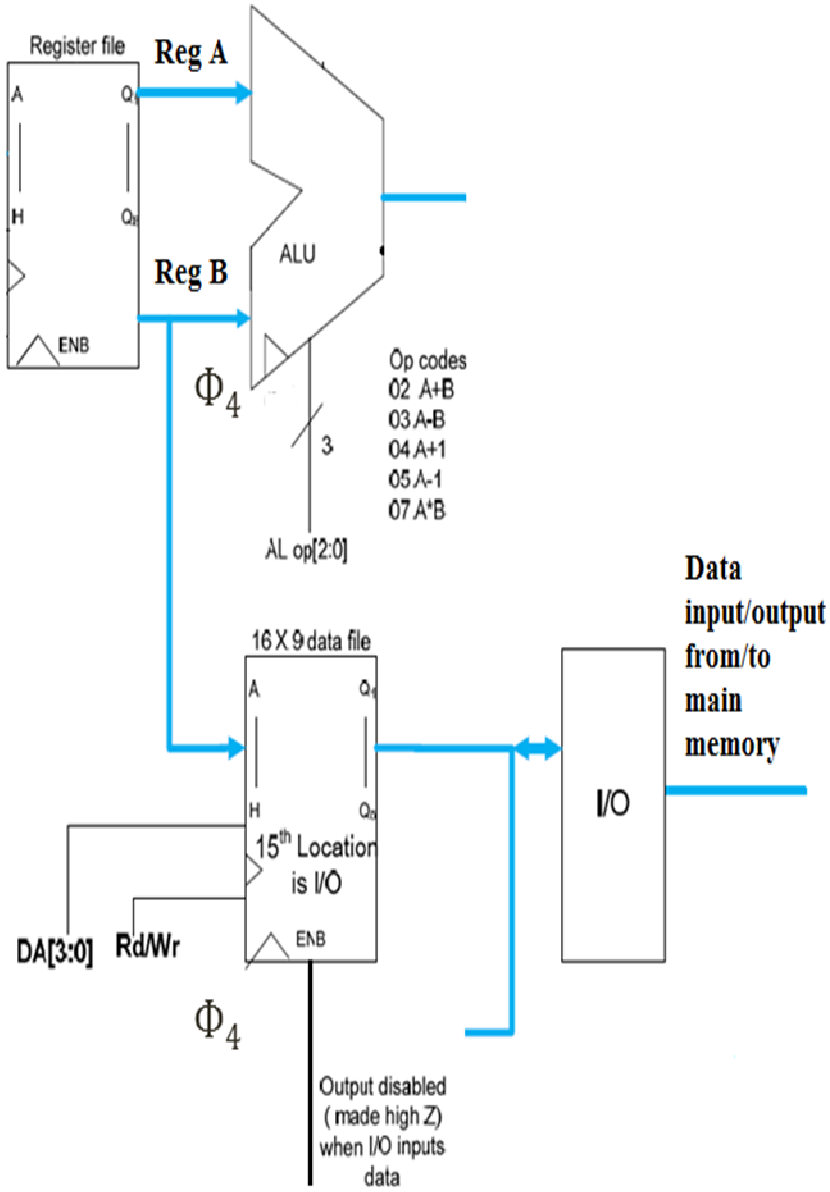
Reg B is also passed to another registry called the data file registry.

Step 4

Use Φ_4 to undertake the instruction in the ALU.

Φ_4 also writes the operand in Reg B to another register called the 'register data file'.

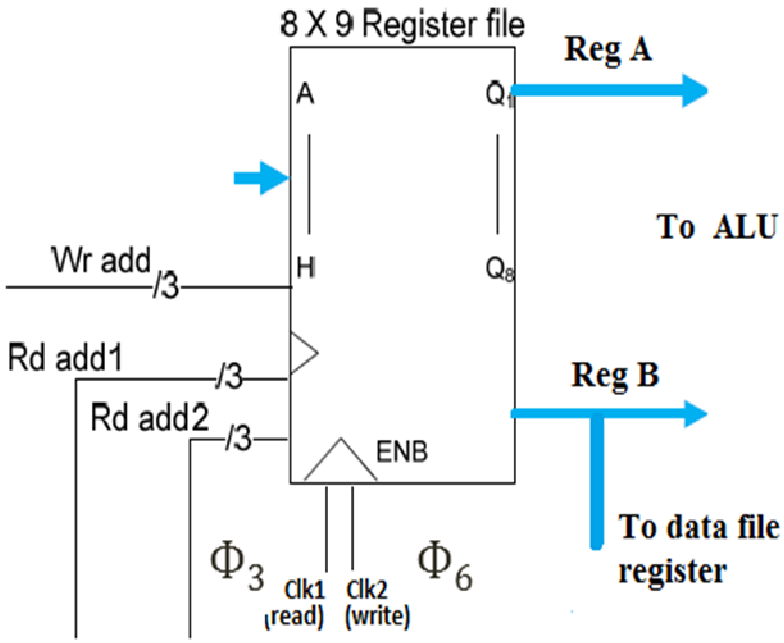
The register data file has two functions, it can either store data or else (controlled by the 15th position in the instruction) it can permit the data from main memory to flow towards the register file and from register file to the data file.



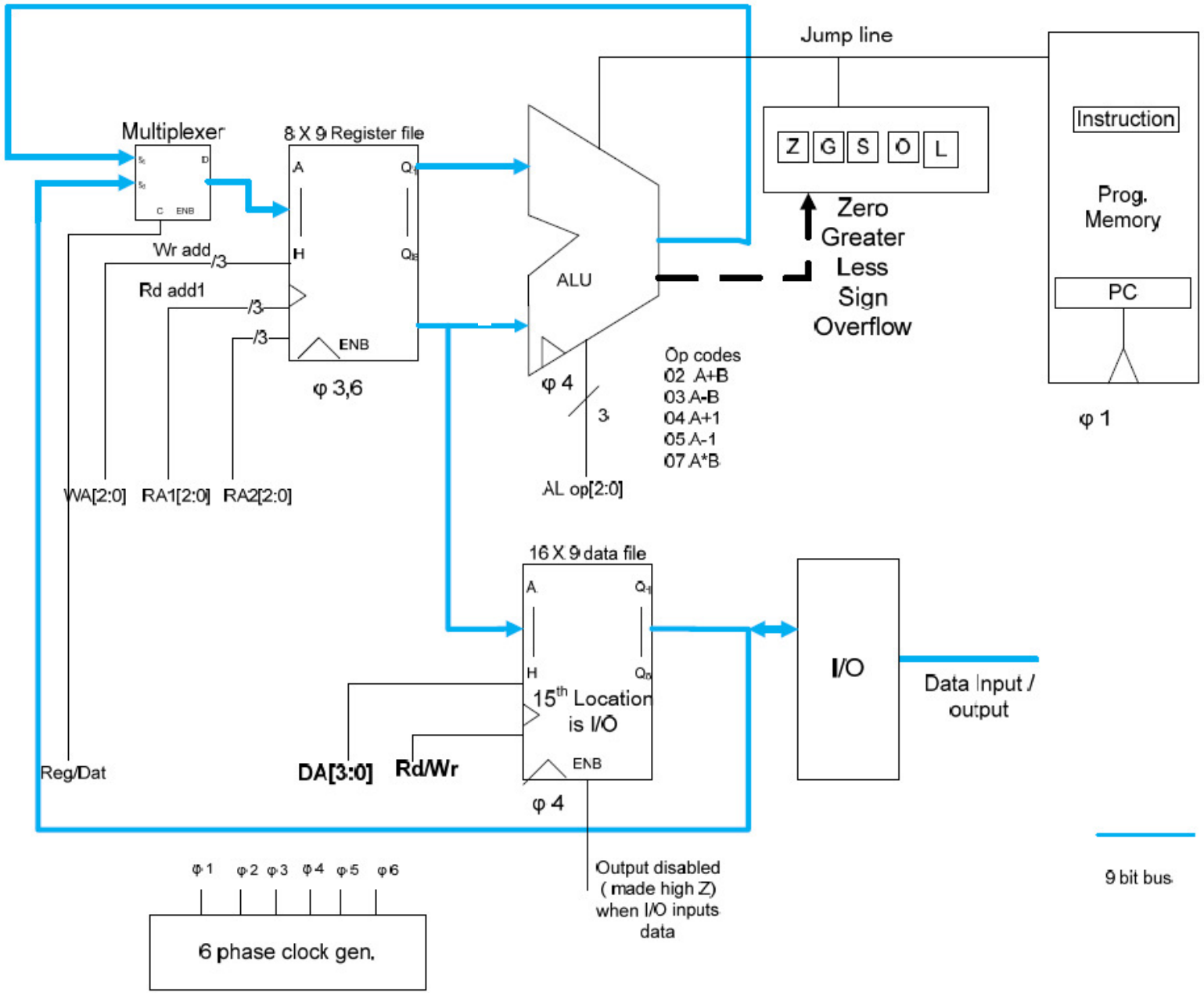
Steps 5 and 6

Use Φ_5 to allow some time for the data to reach the register file.

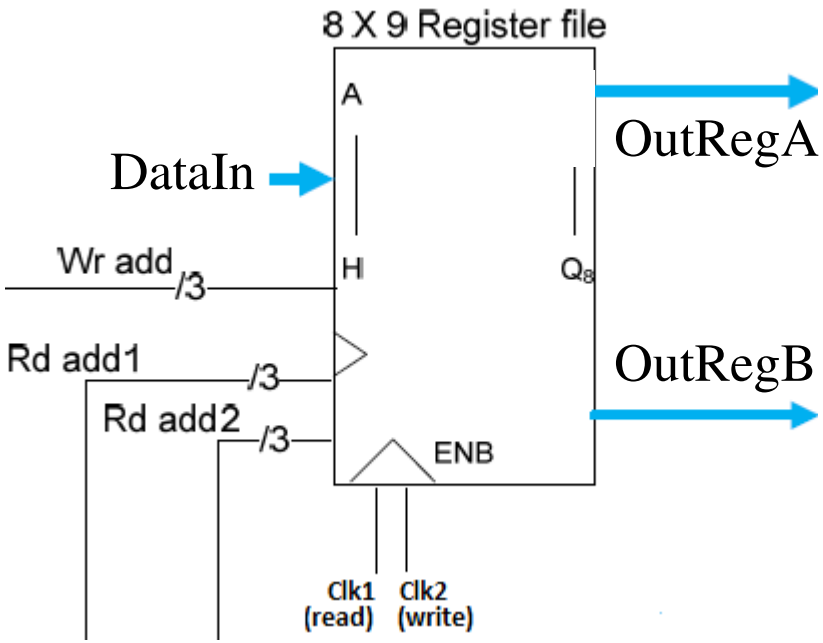
Use Φ_6 to write data to the register file.



Architecture



Register file

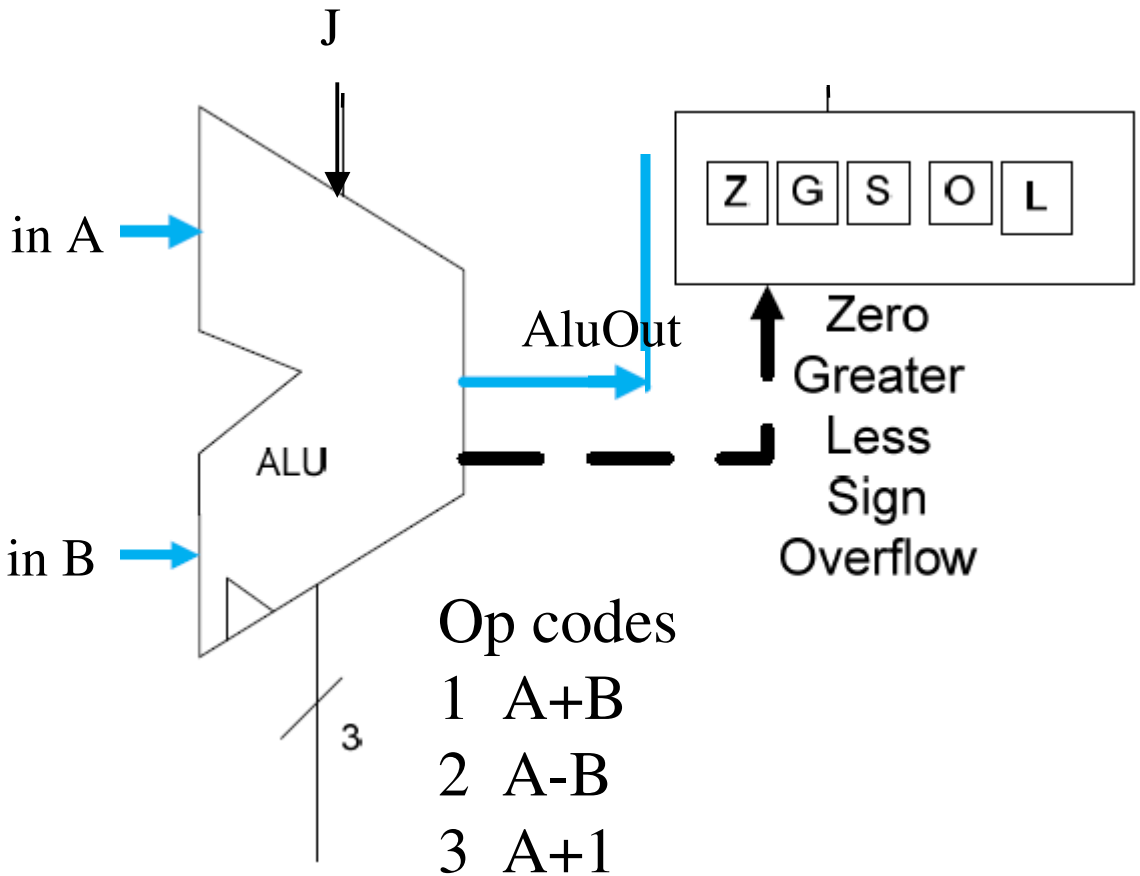


Memory: 8 registers, 9 bits each

One input / Two outputs
One write address (Wr add)
Two read addresses (Rd add1, Rd add2)

Read on clk1
Write on clk2

ALU with status and jump control line



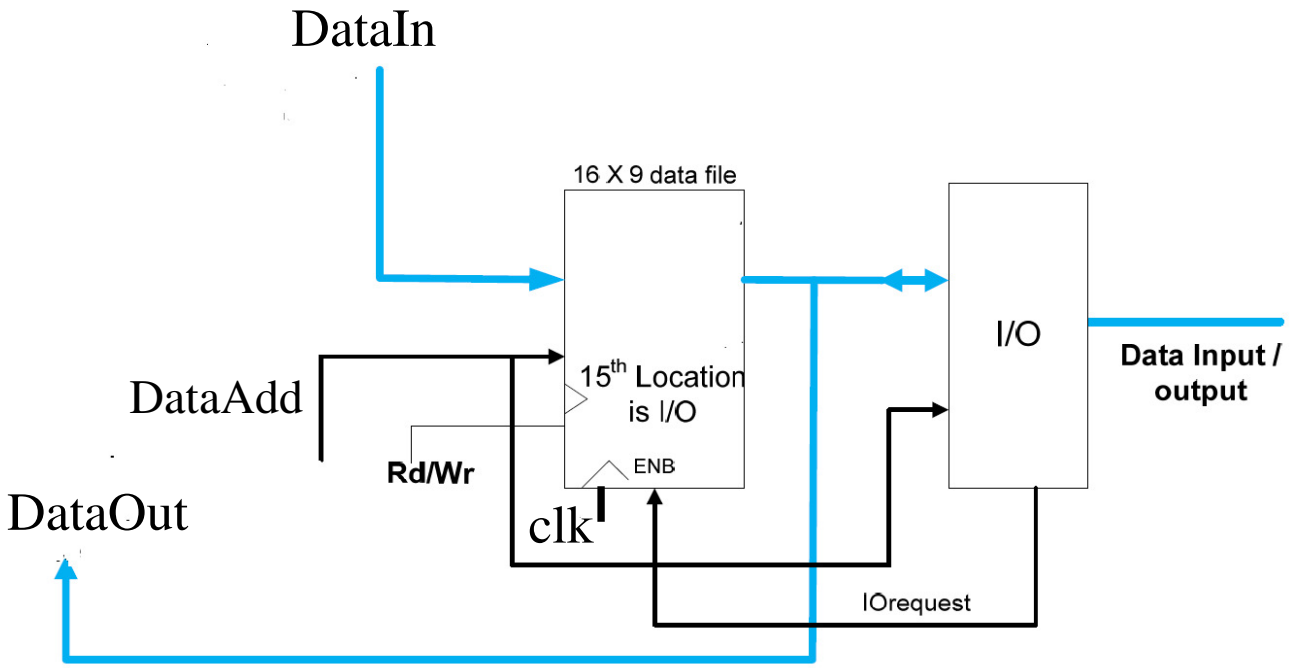
Op codes

1 A+B
2 A-B
3 A+1
4 A-1
5 A*B

Status

if (AluOut==0) Z=1
if (AluOut < 0) L = 1
if (AluOut >0) G = 1
if (AluOut > 511) O = 1
if (AluOut[8]) S=1

Data registers and IO module



The IO module works in conjunction with the data register, i.e when the data address = 0xF the IO request goes high switching the data register port off for IO port reads.

Data registers and IO module: Memory-mapped IO

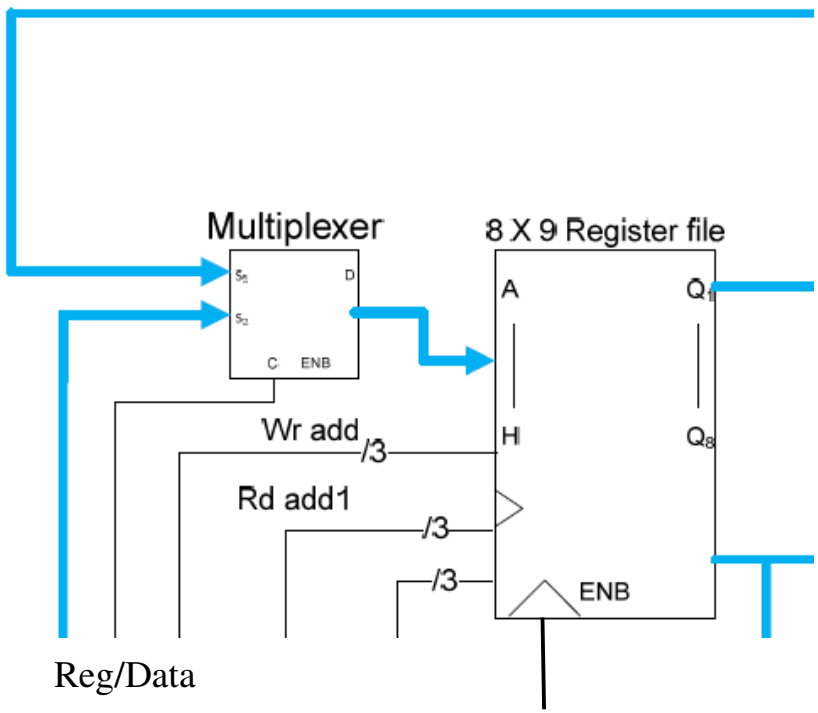
Memory-mapped IO	Port-mapped IO
Same address bus to address memory and I/O devices	Different address spaces for memory and I/O devices
Access to the I/O devices using regular instructions	Uses a special class of CPU instructions to access I/O devices
Most widely used I/O method	x86 Intel microprocessors - IN and OUT instructions

In memory-mapped systems, the I/O device is accessed like it is a part of the memory.This means I/O devices use the same address bus as memory, meaning that CPU can refer to memory *or* the I/O device based on the value of the address. This approach requires isolation in the address space: that is, addresses reserved for I/O should not be available to physical memory.

Port-mapped I/O often uses a special class of CPU instructions specifically for performing I/O. This is found on Intel microprocessors, with the IN and OUT instructions.

As for the advantages and disadvantages: since the peripheral devices are slower than the memory, sharing data and address buses may slow the memory access. On the other hand, by the I/O simplicity memory-mapped systems provide, CPU requires less internal logic and this helps for faster, cheaper, less power consuming CPUs to be implemented.

Bus / Control



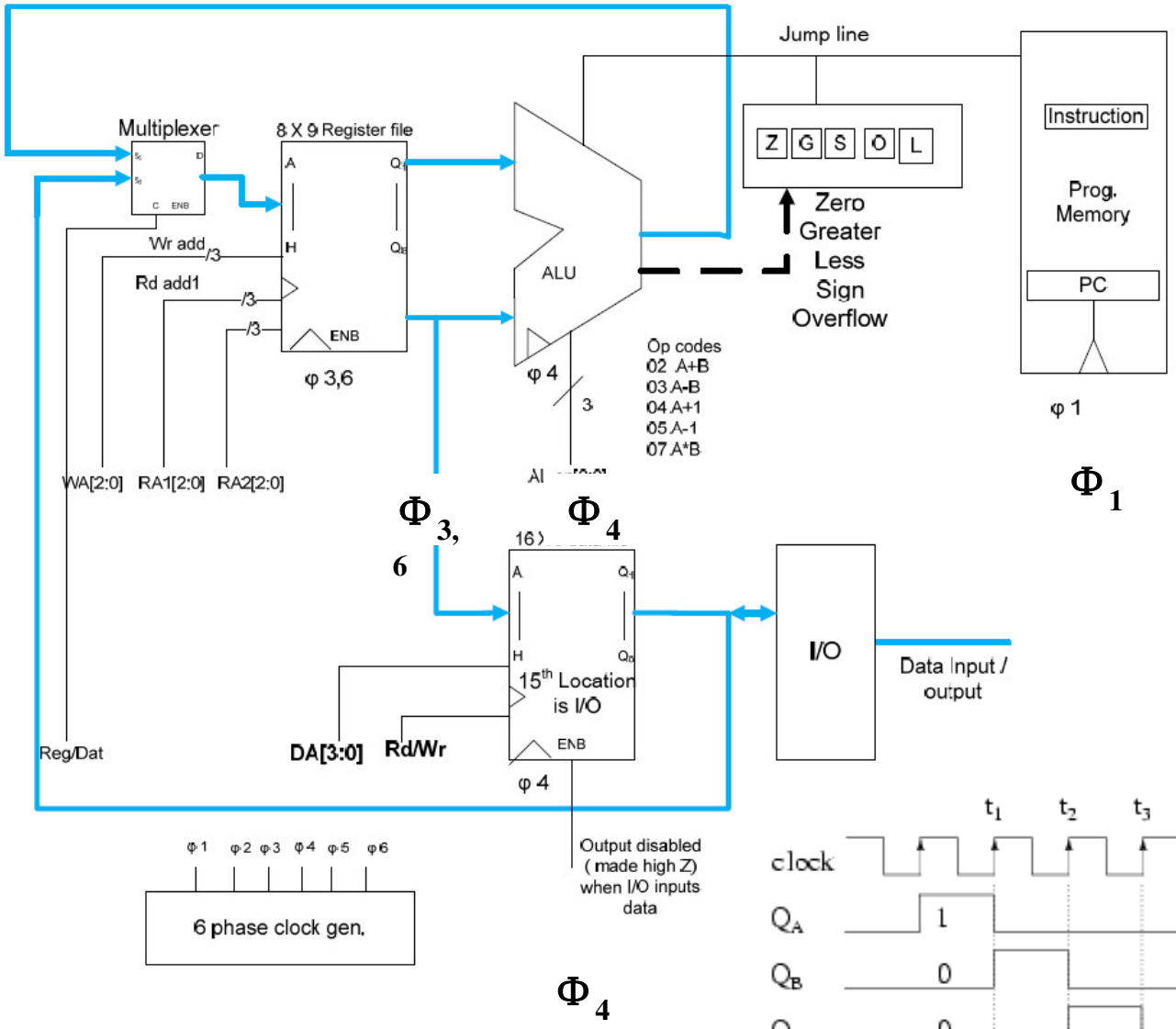
Inputs to the bus are tri-state buffers that only place a signal on the bus when they are enabled

Any number of components can read the bus

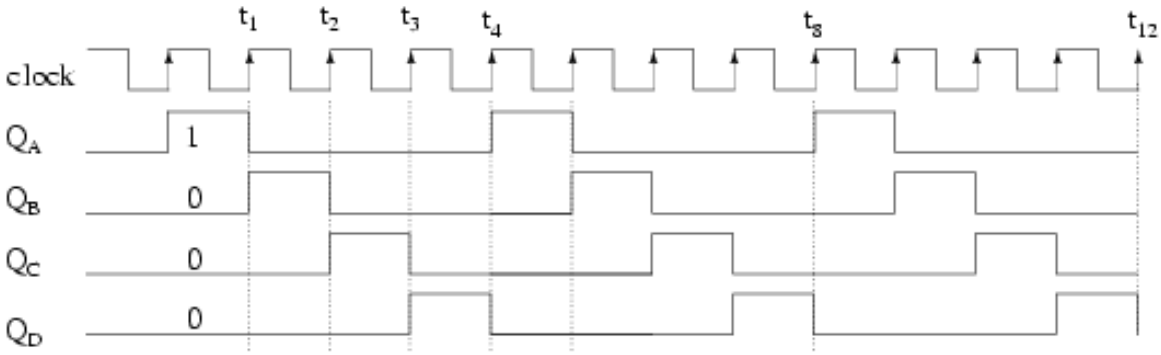
Control unit decides which signal drives the bus and write-enables the destination device

A **tri-state buffer** is similar to a **buffer**, but it adds an additional "enable" input that controls whether the primary input is passed to its output or not. If the "enable" inputs signal is true, the **tri-state buffer** behaves like a normal **buffer**.

Phase shifted clock



Clock signal is distributed on 6 wires, each with non-overlapping pulses
⇒Components are active at different times.
E.g. Register file reads are executed on phase 3, ALU and data file are active on phase 4, while register file write is executed on phase 6



Instruction set: mnemonic codes

LD Rp, Dq - load register Rp,with contents of Dq
LD Dp, Rq - store data in register q in data location p
COP Rp, Rq - copy, Rq into Rp
ADD, Rp, Rq - arithmetically add contents Rp and Rq and store the result in Rp
SUB , Rp,Rq - arithmetically subtract contents Rq from Rp and store the result in Rp
INC, Rp - increment Rp by 1
DEC, Rp - decrement Rp by 1
MUL Rp,Rq - multiply contents of Rp by contents of Rq and store in Rp
AND Rp, Rq – bit AND contents or Rp with contents or Rq and store in Rp
XOR Rp, Rq – bit XOR contents or Rp with contents or Rq and store in Rp
where p is a 3 bit address and q is 4 bit address

Jump instructions
JP Q - always jump relative to current location to new programme location Q. Q is set as a 2s compliment number allowing jumps to occur + 127 forward and -128 back from location
JZ Q - jump relative to location if Zero flag is set. Q is set as in JP instruction
JG Q - jump relative to location if greater than zero flag set. Q is set as in JP instruction
JL Q – jump relative to location if less than zero flag set. Q is set as in JP instruction

Instruction set, encoding: instruction code

Bits	23 :20	19:16	15:12	11:14	7:4	3:0
	IIII,	xWWW,	Reg/data,RRR	DDDD	xRRd/Wx	xxxx

Control field (decimal)	Function
0	No operation
1	Load Reg. with data
2	Add
3	Subtract
4	Increment
5	decrement
6	Copy
7	Multiply
8	Bit AND
9	Bit XOR

Instruction bit field

Bits	23 :20	19:16	15:12	11:14	7:4	3:0
	IIII,	xWWW,	Reg/data,RRR	DDDD	xRRd/Wx	xxxx

Where

- Bits 23:20 is the instruction code taken from table in previous slide,
- bits 18:16 is the write address,
- bit 15 is the Reg/Data(regDat) line,
- bits 14:12 is the read address,
- bits 11:8 DDDD is the data address,
- bit 6 is Register operation,
- bit 5 Read data / Write Data.

All x bits are not currently in use. Four jump instructions are also available

Full instruction set …

Mnemonic Bit positions	IIII 23:20	xWWW 19:16	R/D RRR 15:12	DDDD 11:8	xRRd/Wx 7:4	 3:0	Comments
LD D_N, R_M	0001	0000	0mmm	nnnn	0100	xxxx	Load data register N with contents of register M
LD R_N,D_M	0001	xnnn	0000	mmmm	0110	xxxx	Load register N with contents of data address M
ADD R_N,R_M	0010	xnnn	1mmm	0000	0110	xxxx	$R_N = R_M + R_N$
SUB R_N,R_M	0011	xnnn	1mmm	0000	0110	xxxx	$R_N = R_M - R_N$
INC R_N,	0100	xnnn	1nnnn	0000	0110	xxxx	$R_N = R_N + 1$ (INC R_N, R_M)
DEC R_N	0101	xnnn	1nnnn	0000	0110	xxxx	$R_N = R_N - 1$
COP R_N,R_M	0110	xnnn	1mmm	0000	0110	xxxx	$R_N = R_M$
MUL R_N,R_M	0111	xnnn	1mmm	0000	0110	xxxx	$R_N = R_N * R_M$
AND R_N R_M	1000	xnnn	1mmm	0000	0110	xxxx	$R_N = R_N$ bit And R_M
XOR R_N R_M	1001	xnnn	1mmm	0000	0110	xxxx	$R_N = R_N$ bit XOR R_M
JP	1010	QQQQ	QQQQ	0000	0010	xxxx	Jump relative to current location to Q where Q is a 2'compliment numbe r + 127 forward, -128 back. Bits in this field are ignored for jumps
JZ	1011	QQQQ	QQQQ	0000	0010	xxxx	Jump if Zero flag set to relative Q location
JG	1100	QQQQ	QQQQ	0000	0010	xxxx	Jump if Greater than flag set to relative Q location
JL	1101	QQQQ	QQQQ	0000	0010	xxxx	Jump if less flag set to relative Q location

M and N are register numbers with legal values 0 – 7 and m and n are the bit representations of M and N. N is the destination register address, M is the source address. The xWWW column is the destination register and RRR is the source register for both register to register and data transfers. Bit 15 is his high for a register transfer and low for a data transferFor the load instruction Data to register D is source and xWWW is the destination. For load register to data RRR is the source register and D is the destination register. The column labelled DDDD is the data register address The next column consists of 4bits, R is set for any register operation, Rd/W is a single bit field for the data register that is 1 when read and 0 when written. X are unused bits and their default value should be set to 0

Instruction execution

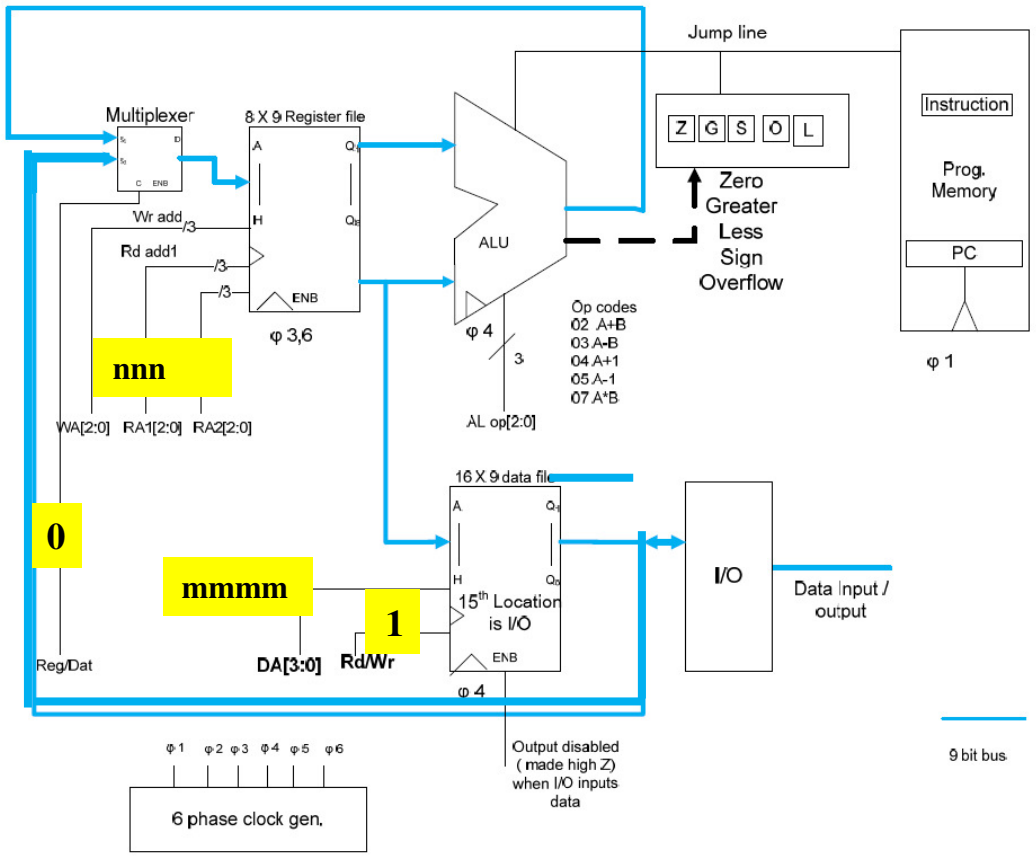
For the program to execute properly, the controller must set up a number of control signals, based on the instruction it receives from program memory

WA	RA	DA	Rd/Wr	Reg	Reg/Data	ALU_control
Address of register to write	Address of register to read	Address of data register	If 1 read data else write data at address DDDD	Set if any reg change required	If set register file input is ALU result else data file	2 = A+ B 3= A- B etc

Note: for operations with 2 registers operands, e.g. ADD Rm, Rn, RA is used as the address of one operand (m), while WA is used as address of the other operand (n)

Instruction execution example: LD Rn, Dm

Mnemonic Bit positions	IIII 23:20	xWWW 19:16	R/D RRR 15:12	DDDD 11:8	xRRd/Wx 7:4	 3:0	Comments
LD R_N,D_M	0001	xnnn	00000	mmmm	0110	xxxx	Load register N with contents of data address M



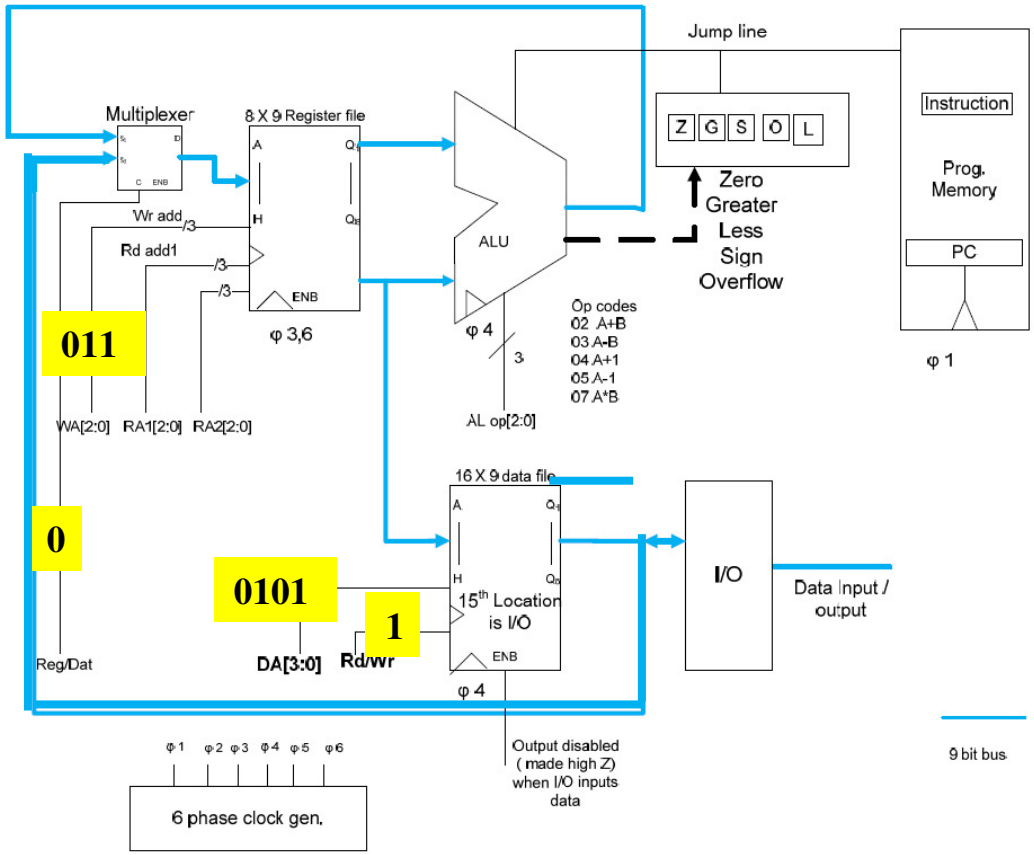
Encoding of instruction

Corresponding control signals and datapaths when instruction is executed

Instruction execution example: LD R3, D5

Mnemonic Bit positions	IIII 23:20	xWWW 19:16	R/D RR R 15:12	DDDD 11:8	xRRd/Wx 7:4	 3:0	Comments
LD R_N,D_M	0001	xnnn	00000	mmmm	0110	xxxx	Load register N with contents of data address M

Encoding of instruction



…

Phase 4:

Read register 5 from data file

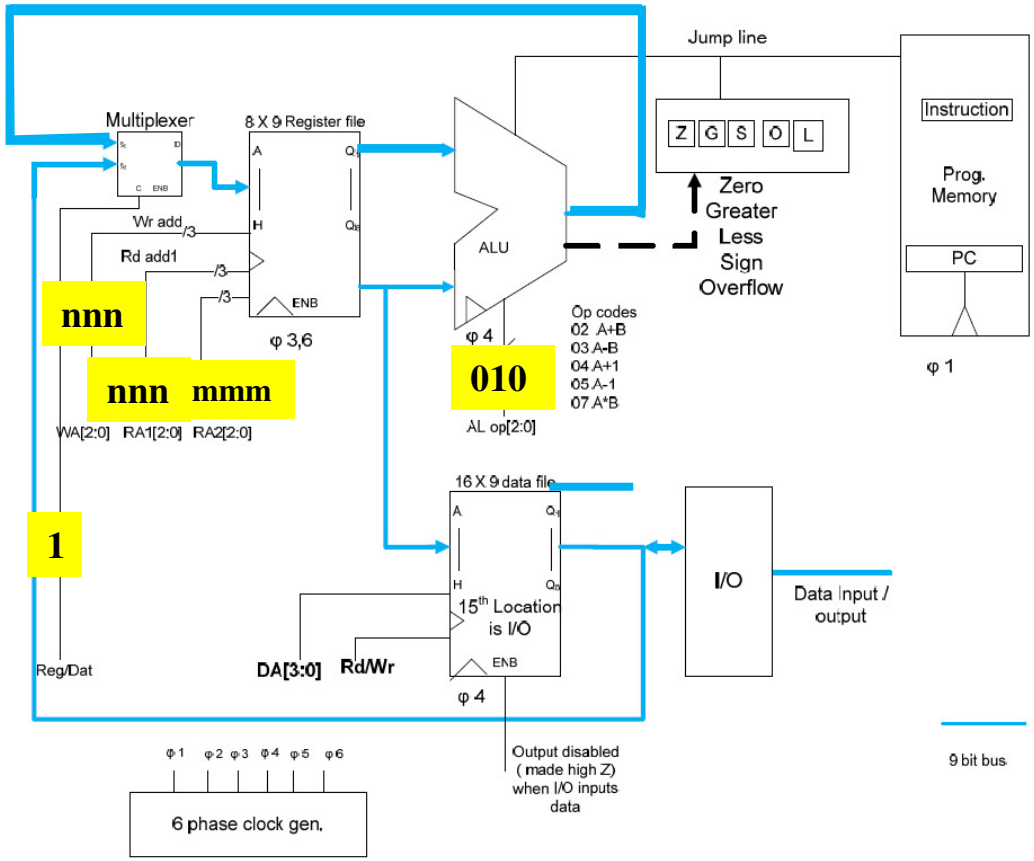
Phase 6:

Write register 3 into registers file

Instruction execution example: Add Rm, Rn

Mnemonic Bit positions	IIII 23:20	xWWW 19:16	R/D RRR 15:12	DDDD 11:8	xRRd/Wx 7:4	 3:0	Comments
ADD R_N,R_M	0010	xnnn	1mmm	0000	0110	xxxx	$R_N = R_M + R_N$

Encoding of instruction

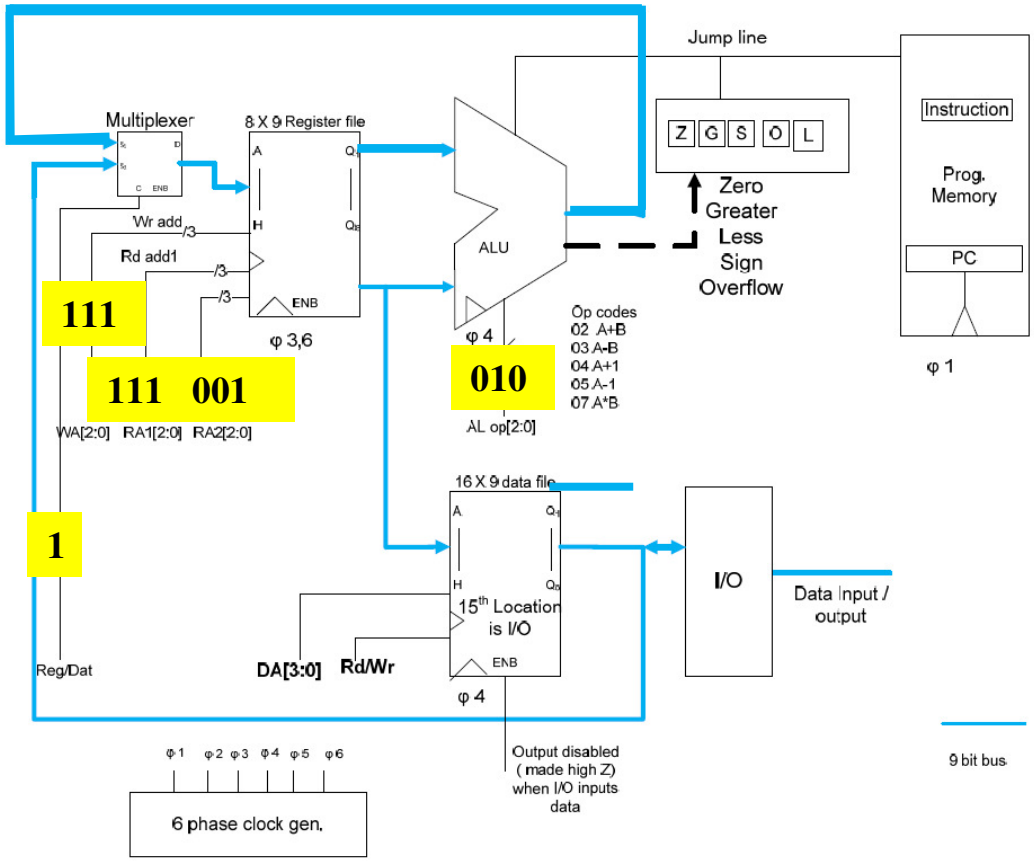


Corresponding control signals and datapaths when instruction is executed

Instruction execution example: Add R7, R1

Mnemonic Bit positions	IIII 23:20	xWWW 19:16	R/D RRR 15:12	DDDD 11:8	xRRd/Wx 7:4	 3:0	Comments
ADD R_N,R_M	0010	xnnn	1mmm	0000	0110	xxxx	$R_N = R_M + R_N$

Encoding of instruction



…

Phase 3:

read register 7 and 1 from reg. file

Phase 4:

sum

Phase 6:

store result in register 7

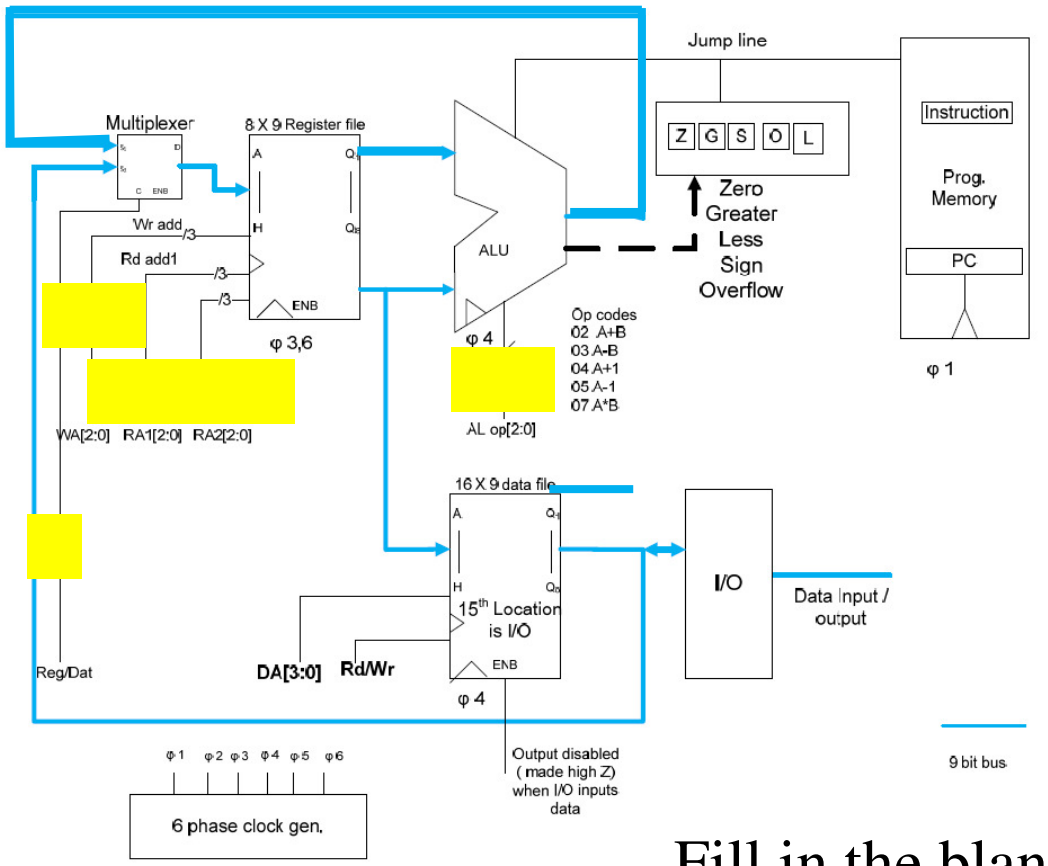
Example program

For example, if the program stored in memory consists of the five instructions below:
(Data 1 and Data 2 have been previously loaded with 18d and 20d)

```
L1: 100160 // 0001 0000 0000 0001 0110 0000 // load R0, D1 R0 = 18d
    110240 // 0001 0001 0000 0010 0110 0000 // load R1, D2 R1 = 20d
    209060 // 0010 0000 1001 0000 0110 0000 // R0= R0 + R1 R0 = 38d
    419060 // 0100 0001 1001 0000 0110 0000 // R1 = R1 + 1 R1 = 21d
    AFC000 // 1010 1111 1100 0000 0010 0000 // jump  -  4 to the beginning,…
                                             // …PC= address of L1
```

Class exercise: COP R_N, R_M

Mnemonic Bit positions	IIII 23:20	xWWW 19:16	R/D RRR 15:12	DDDD 11:8	xRRd/Wx 7:4	 3:0	Comments
COP R_N,R_M	0110	xnnn	1mmm	0000	0110	xxxx	$R_N = R_M$



Fill in the blanks

…

Phase 3:

read register and from reg. file

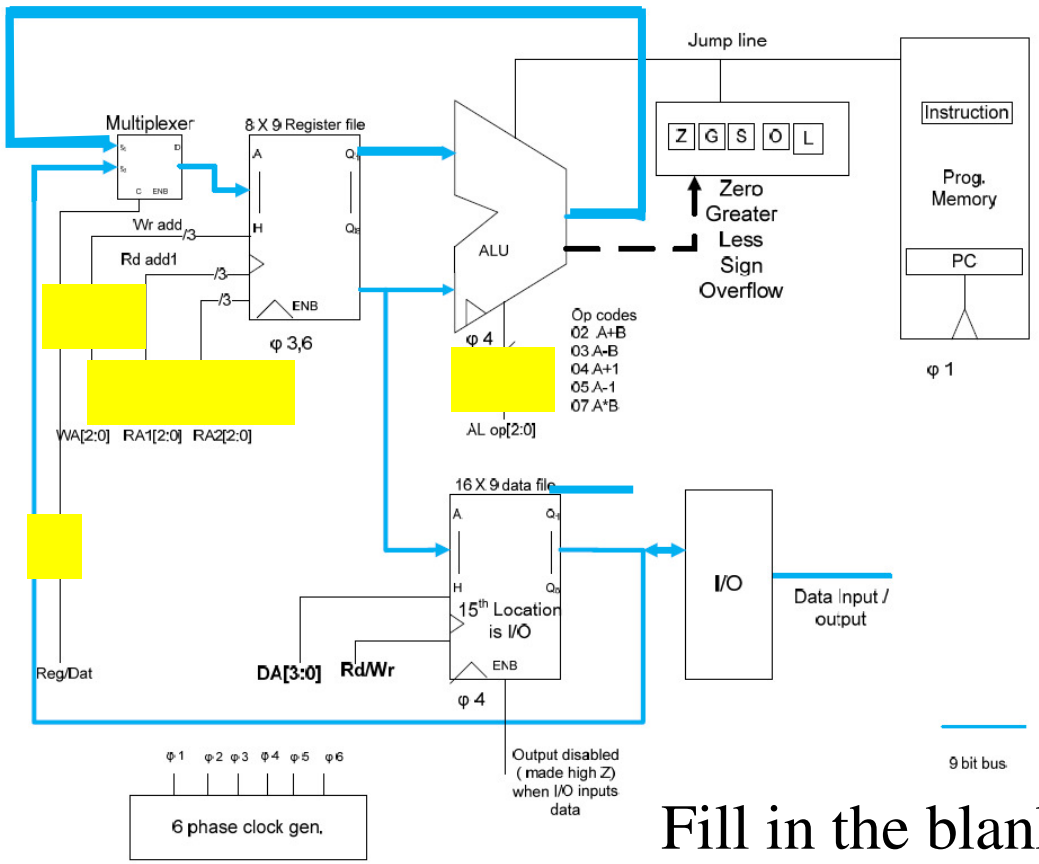
Phase 4:

Phase 6:

store result in register

Class exercise: INC R_N

Mnemonic Bit positions	IIII 23:20	xWWW 19:16	R/D RRR 15:12	DDDD 11:8	xRRd/Wx 7:4	 3:0	Comments
INC R_N,	0100	xnnn	1nnnn	0000	0110	xxxx	$R_N = R_N + 1$ (INC R_N, R_M)



Fill in the blanks

Encoding of instruction

…

Phase 3:

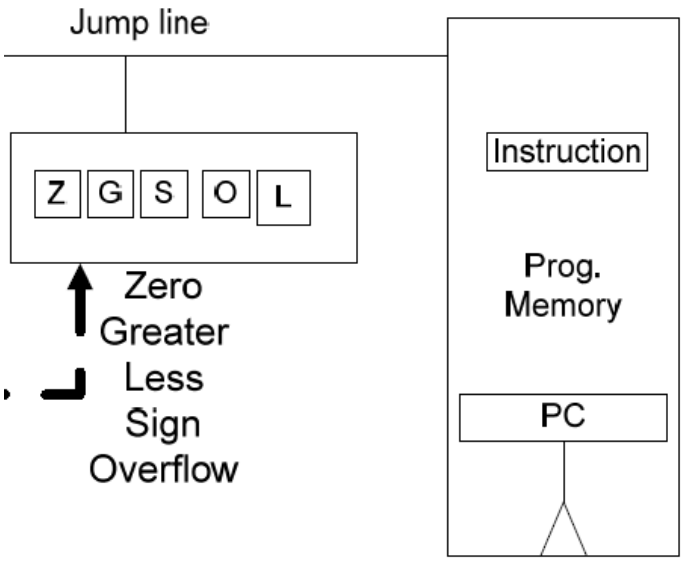
read register and from reg. file

Phase 4:

Phase 6:

store result in register

Program counter, instruction, program memory



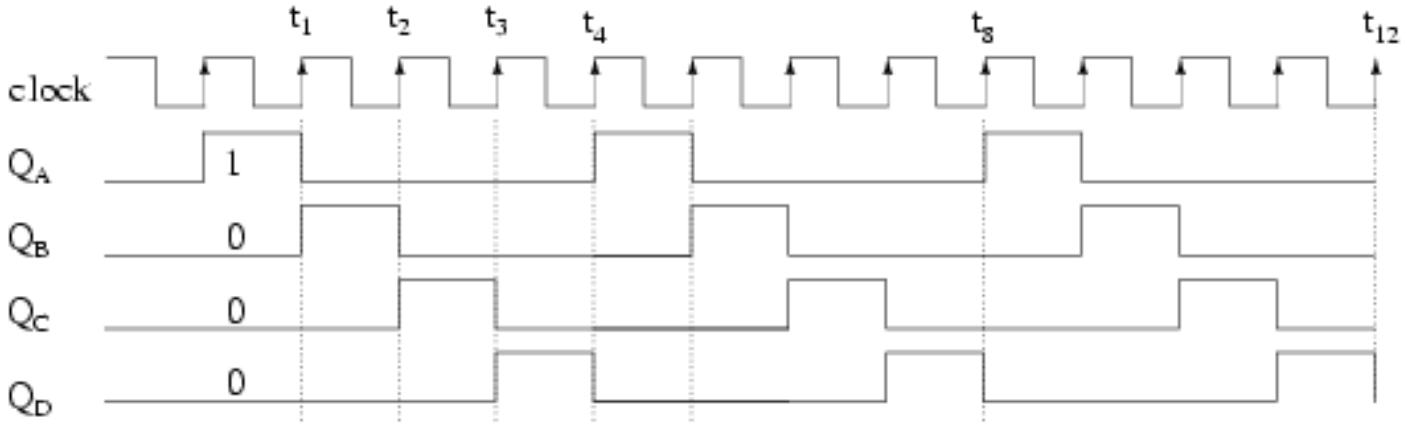
Programs are stored as sequence of instructions in Program Memory

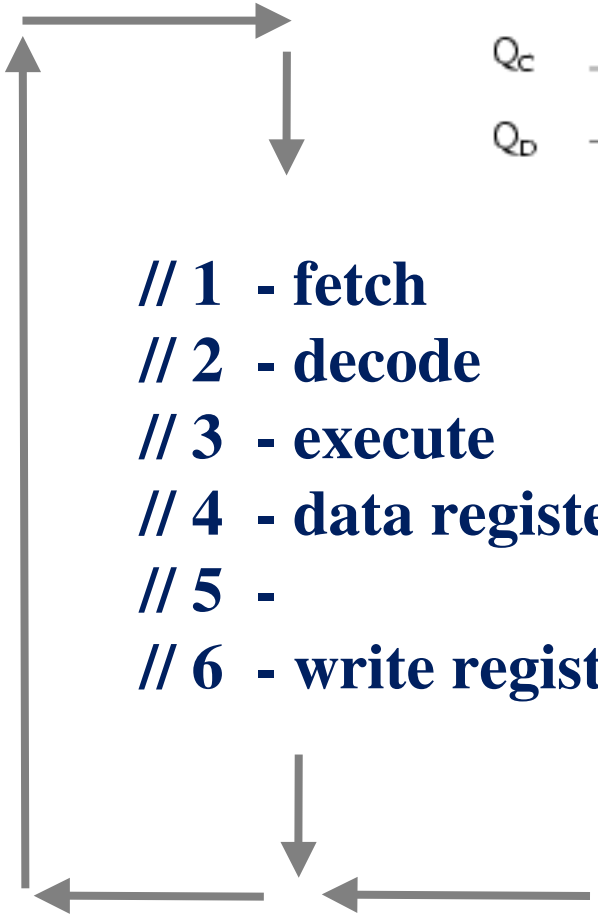
PC is the program counter, pointing the controller to the current instruction

At power on, PC is initialised to zero, therefore the memory sees PC = 0, and sends the first 24 - bit instruction to the controller

It is then the function of the controller to read this 24 - bit instruction, and set up control signals that control the data path, as necessary

Algorithm of the Controller





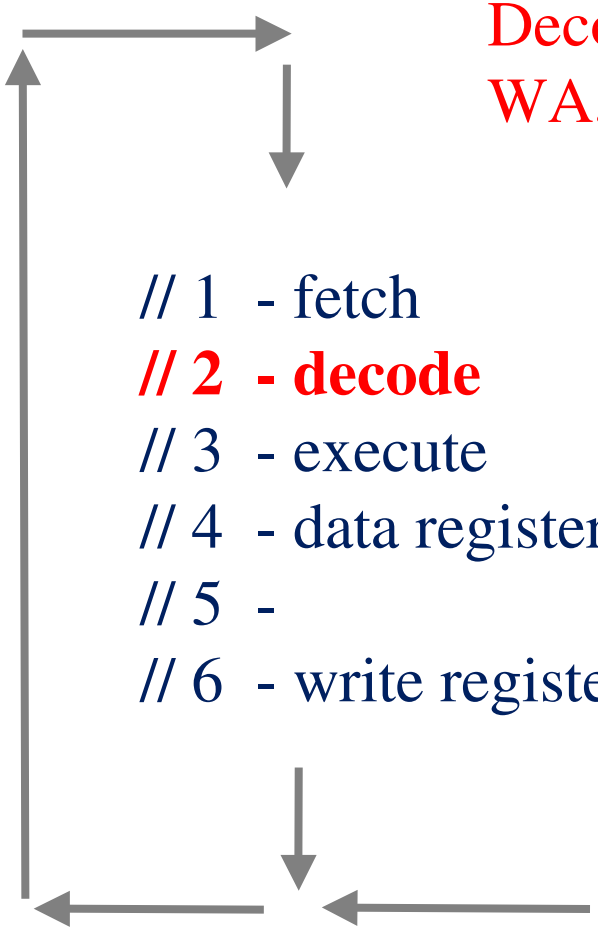
// 1 - fetch
// 2 - decode
// 3 - execute
// 4 - data registers operation / ALU operation
// 5 -
// 6 - write registers

Algorithm of the Controller

Load next instruction (at address stored in PC) from
Program memory into Instruction Register (IR)
If jump condition, set PC accordingly,
else (default) PC ← PC + 1

// 1 - fetch
// 2 - decode
// 3 - execute
// 4 - data registers operation / alu operation
// 5 -
// 6 - write registers

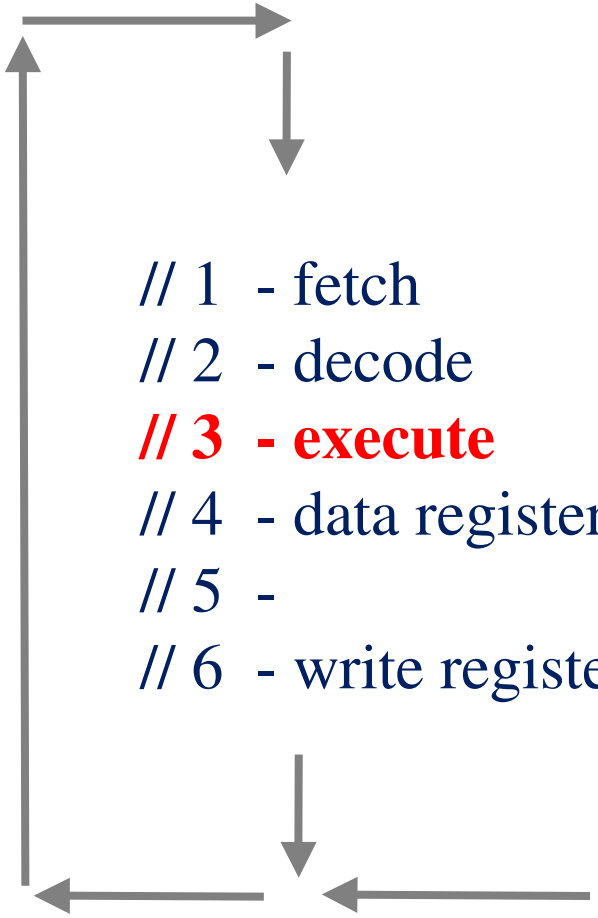
Algorithm of the Controller



Decode the instructions, e.g. read the op code, identify WA, RA, regDat, DA, …

// 1 - fetch
// 2 - decode
// 3 - execute
// 4 - data registers operation / alu operation
// 5 -
// 6 - write registers

Algorithm of the Controller



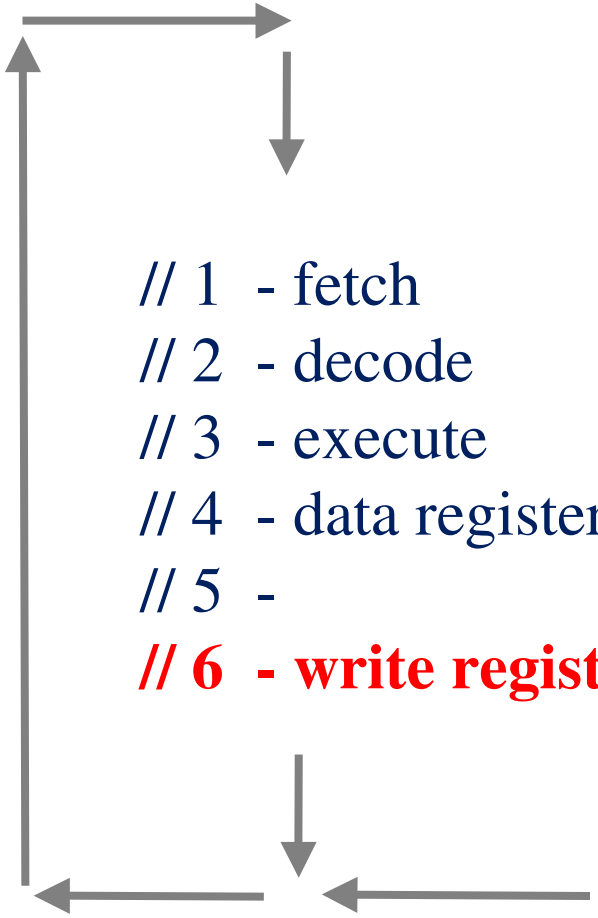
// 1 - fetch
// 2 - decode
// 3 - execute Set ALU operation, read register file
// 4 - data registers operation / alu operation
// 5 -
// 6 - write registers

Algorithm of the Controller

// 1 - fetch
// 2 - decode
// 3 - execute
// 4 - data registers operation / ALU operation
// 5 -
// 6 - write registers

Data registers op / ALU operation

Algorithm of the Controller



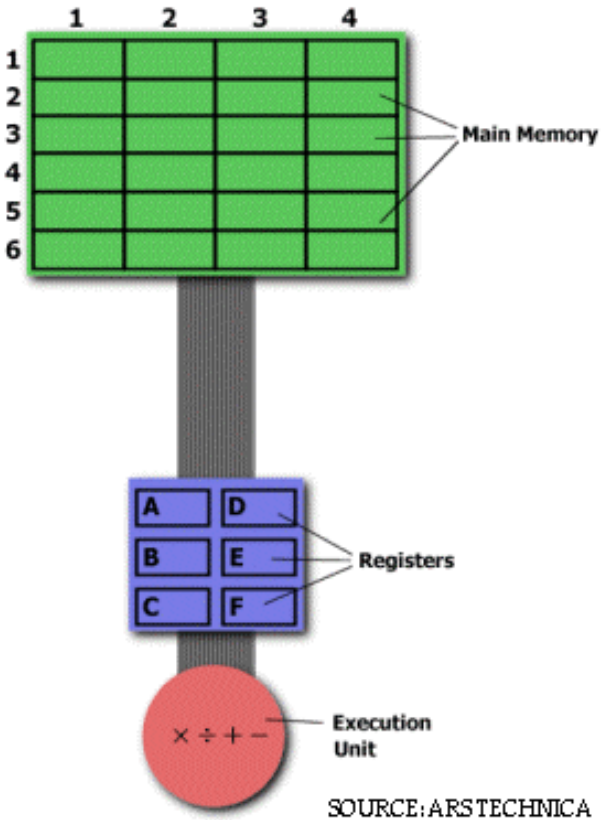
// 1 - fetch
// 2 - decode
// 3 - execute
// 4 - data registers operation / alu operation
// 5 -
// 6 - write registers

Store results

RISC and CISC

The **CISC** (Complex Instruction Set Computer) architecture tries to reduce the number of Instructions that a program has. This is done by combining many simple instructions into a single complex one.

The **RISC** (Reduced Instruction Set Computer) architecture has more instructions, but they reduce the number of cycles that an instruction takes to perform. Generally, a single instruction in a RISC machine will take only one CPU cycle.



CISC

```
MULT 2:3, 5:2
```

Emphasis on hardware

Includes multi-clock complex instructions

RISC

```
LOAD A, 2:3
LOAD B, 5:2
PROD A, B
STORE 2:3, A
```

Emphasis on software

Single--clock reduced instructions only

Because all of the instructions execute in a uniform amount of time (i.e. one clock), pipelining is possible.

The Heriot-Watt machine would be thought as being closer to the RISC idea than the simple data path design.

CISC – Complex Instruction Set Computer

- Memory in those days was expensive
- Bigger program->more storage->more money
- Hence needed to *reduce* the number of instructions per program
- Number of instructions are reduced by having *multiple operations* within a single instruction
- Multiple operations lead to many different kinds of instructions that access memory
- In turn making instruction length variable and fetch-decode-execute time unpredictable – making it more complex
- Thus hardware handles the complexity
- Example: x86 ISA

CISC – Complex Instruction Set Computer

- Microprogramming – complex instructions are split into a series of simpler instructions
- When a complex instruction is executed, the CPU executes a small microprogram stored in a control memory
- This simplifies design of processor and allows the addition of new complex instructions

RISC– Reduced Instruction Set Computer

- Attempt to make architecture simpler
- Reduced number of instructions
- Make them all the same format if possible
- Reduce the number of memory accesses required by increasing the number of registers
- Reduce the number of addressing modes
- Allow pipelining of instructions

RISC and CISC

There is really no “better” architecture, each has its own advantages and disadvantages that make it useful in different applications.

The CPU industry is divided between two very big players backing one of the either techniques. While many **Intel** CPU’s are CISC architecture based, all **Apple** CPUs and **ARM** devices have RISC architectures under the hood.

CISC is most often used in automation devices whereas RISC is used in video and image processing applications.