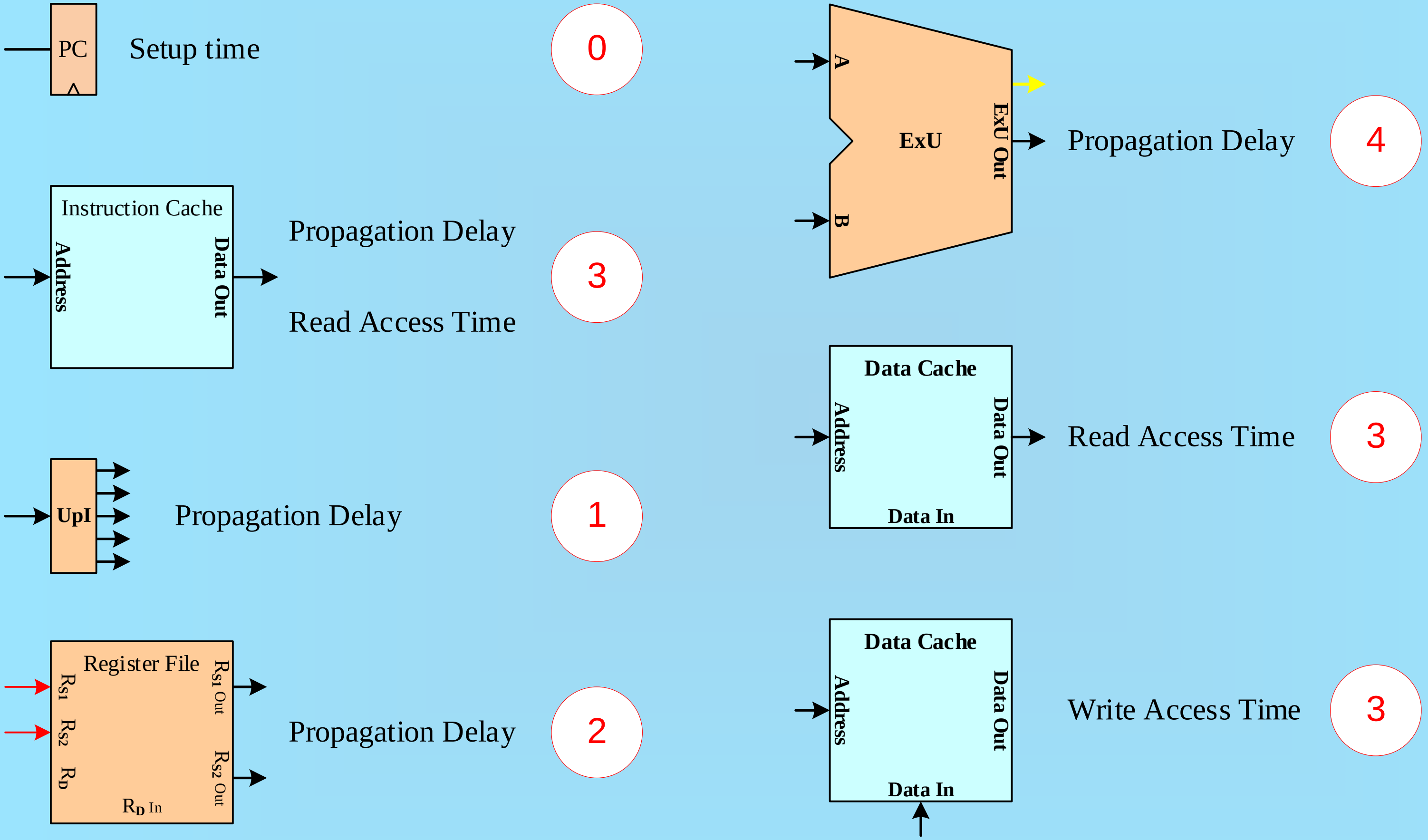
Lecture 11/12

A MULTI-CYCLE RISC-V PROCESSOR

Control Word Table

Timing Analysis of a Single-Cycle Datapath

Consider the approximate worst-case timing of the major elements of the datapath.



Setup time 0

Propagation Delay / Read Access Time 3

Propagation Delay 1

Propagation Delay 2

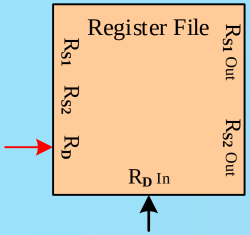
Propagation Delay 4

Read Access Time 3

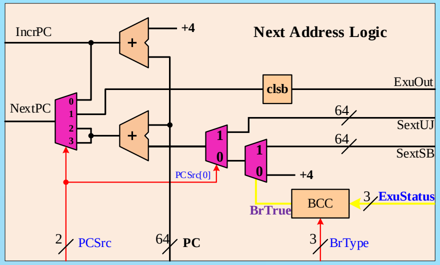
Write Access Time 3

Assume that the setup time for the PC is small in comparison to the ExU & NextAddress elements. The PC setup time is included into their worst-case delay specification.

Timing Analysis of a Single-Cycle Datapath

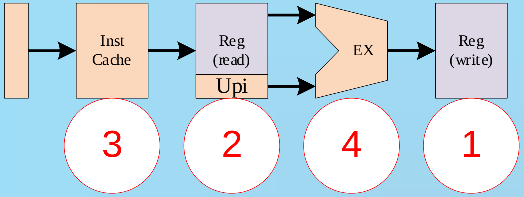


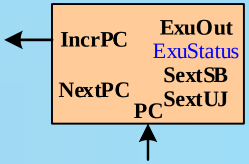
Setup time 1



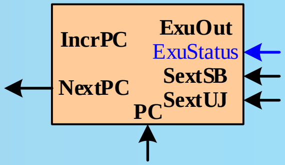
Execute Class Instruction

44%, 10 ns





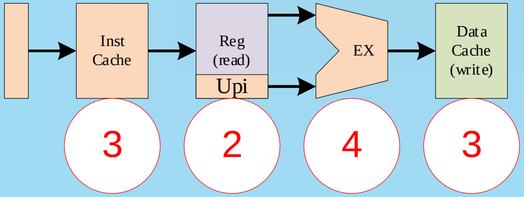
Propagation Delay 1

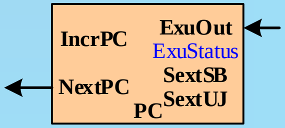


Propagation Delay 2

Memory Class Instruction (store)

12%, 12 ns

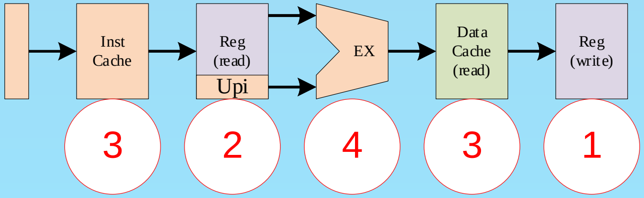




Propagation Delay 1

Memory Class Instruction (load)

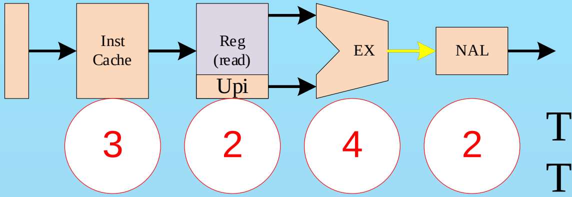
24%, 13 ns



Determining the Instruction Rate. (Throughput)

Flow Control Class Instruction (branch)

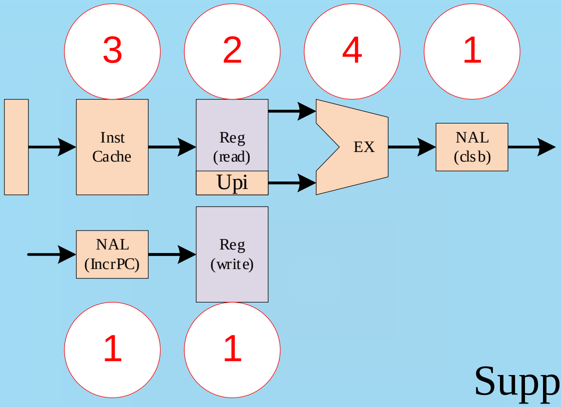
18%, 11 ns



The processor can be clocked at 13 ns
Thus we have on average
13.0 ns per instruction.
or the throughput (reciprocal) is
76.92 MIPS

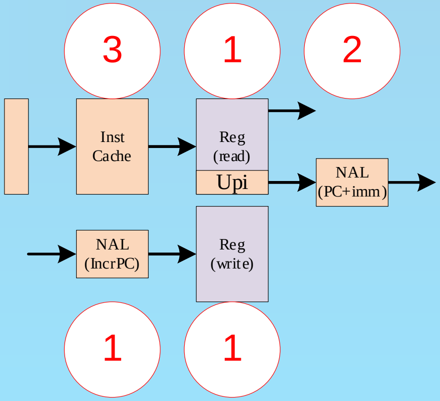
Flow Control Class Instruction (jalr)

1%, 10 ns



Flow Control Class Instruction (jal)

1%, 6 ns



Suppose that 100 instructions execute.

44*10 = 440 ns
12*12 = 144 ns
24*13 = 312 ns
18*11 = 198 ns
1*6 = 6 ns
1*10 = 10 ns

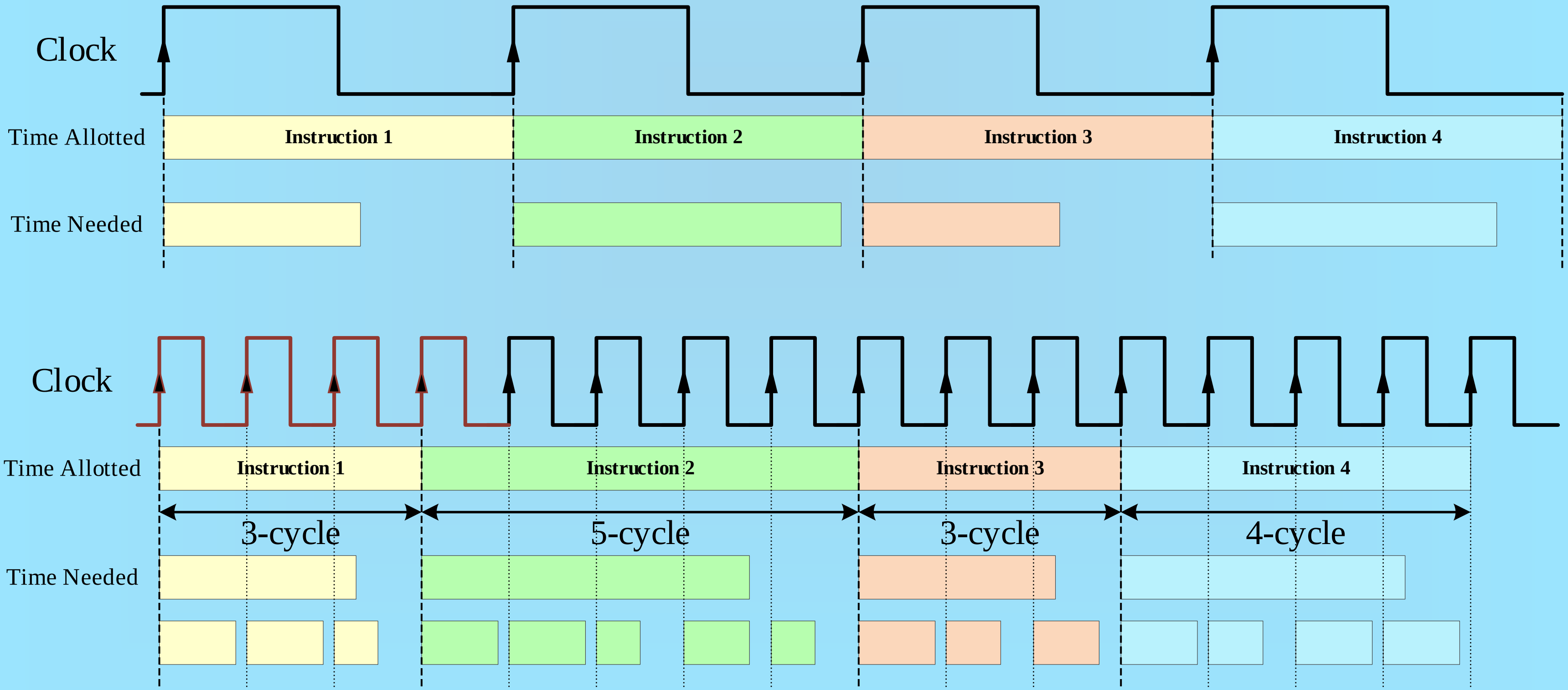
Thus 100 instructions takes 1110 ns
Ideally, we have on average
11.1 ns per instruction.
or the throughput (reciprocal) is
90.09 MIPS

Multi-Cycle Operation

The **RV64I ISA** does not specify that instructions must complete in a single clock cycle.
We are free to design a processor whereby individual instructions have individual cycle counts.

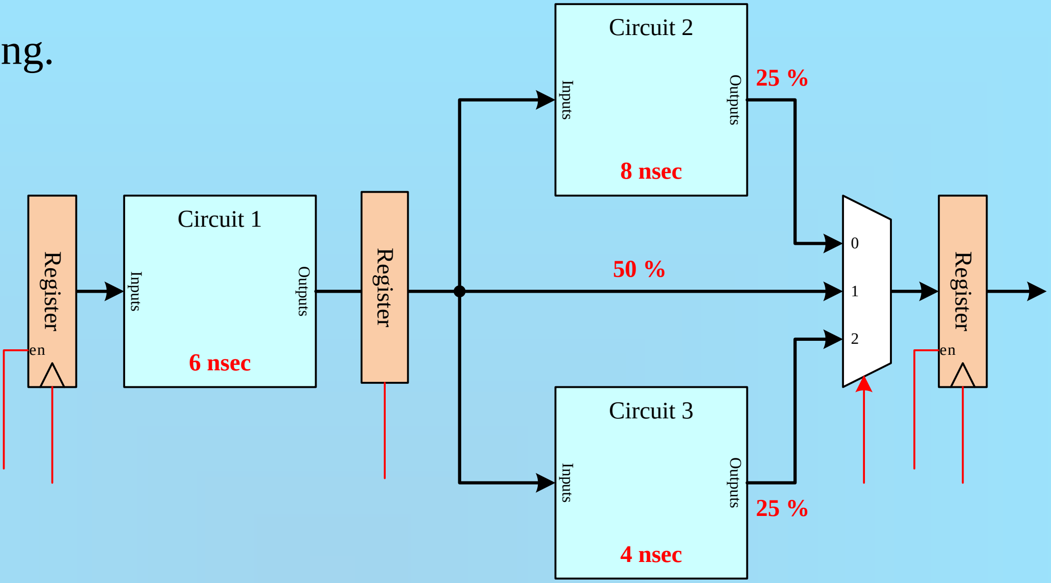
If we introduce additional temporary registers, *invisible to the programmer*, we can hold intermediate results and allow instructions to process over more than one clock cycle. Furthermore;

- it is not necessary that all instruction have the same cycle count, and,
- hardware resources (such as an adder) may be re-used in different cycles for different purposes.



MultiCycle Instructions

Ignore register and MUX timing.



Single-Cycle Operation:

Clock must be (6+8)=14 ns, **on average 5 ns is wasted** every 14 ns.

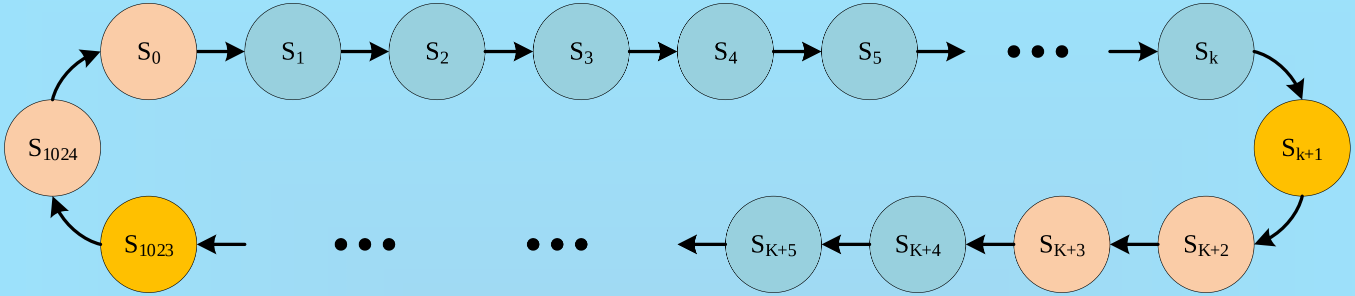
Multi-Cycle operation: *Assume that the new register is a transparent latch*.

The clock period is now 8 ns and the input/output registers are only enabled when it is time for a new calculation. (instruction)

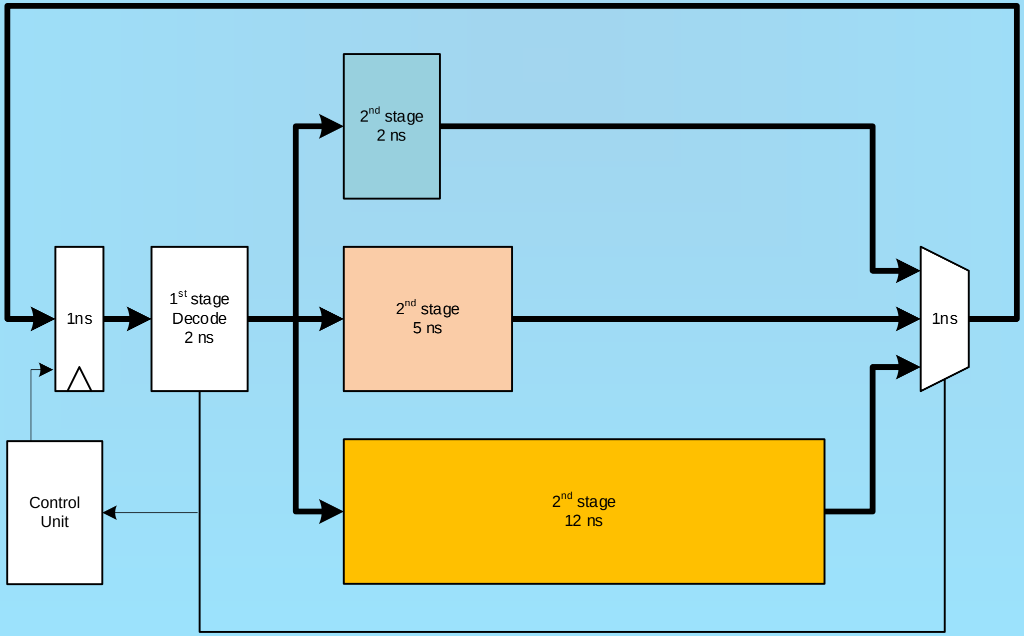
on average, 3 ns is wasted per instruction, **1.5 cycles per instruction**,
thus, on average, **2 ns is wasted every 8 ns**.

Circuit**1** only :	One cycle needed	– occurs 50%, wastes 2 ns.
Circuit**1** and Circuit**2**:	Two cycles needed	– occurs 25%, wastes 2 ns
Circuit**1** and Circuit**3**:	Two cycles needed	– occurs 25%, wastes 6 ns

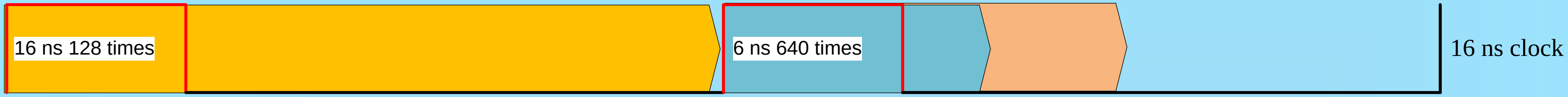
Multi-Cycle Principle



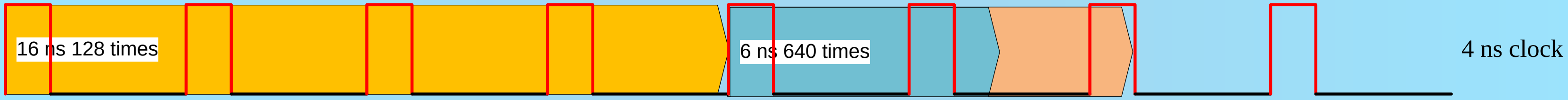
1024 instructions - **128 long**, **256 mid** and **640 short**.



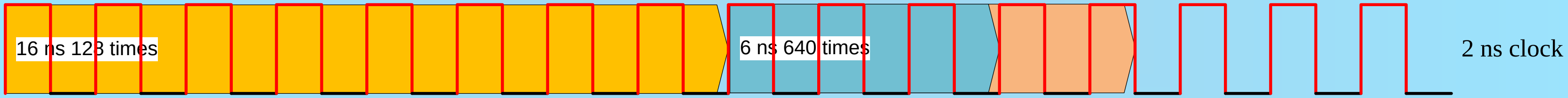
Multi-Cycle Principle



1*128 + **1***256 + **1***640 = 1024 clock cycles, T = 16*1024 nsec



4*128 + **3***256 + **2***640 = 2560 clock cycles, T = 10*1024 nsec



8*128 + **5***256 + **3***640 = 4224 clock cycles, T = 8.25*1024 nsec

Overlapping Decode and Register Access

The first two cycles of every instruction will always be **Instruction Fetch** and **Instruction Decode**. The Control Unit cannot know the total number of required cycles for the current instruction until the instruction has been fetched and decoded.

- Thus ALL instructions must begin with 2 **identical** cycles, **FETCH** and **DECODE**.

After the first clock cycle, the bits of the instruction are stable. The ***Opcode/Funct7/Funct3*** bits and the Register Address bits are available after the first clock cycle.

The **RV64I ISA** instruction formats allow instruction decoding and register read access to occur in parallel.
Both, the single-cycle and the multi-cycle processor designs may perform the Instruction Decoding in **parallel** with the **Register Read Access**.

Sometimes instructions such as `jal` do not use values from the registers. This point is irrelevant since such instructions may simply ignore the register output values.

Resource Sharing in a Multi-Cycle Datapath

The main differences in the new Datapath arise due to the fact that a single resource may now be used more than once in different cycles.

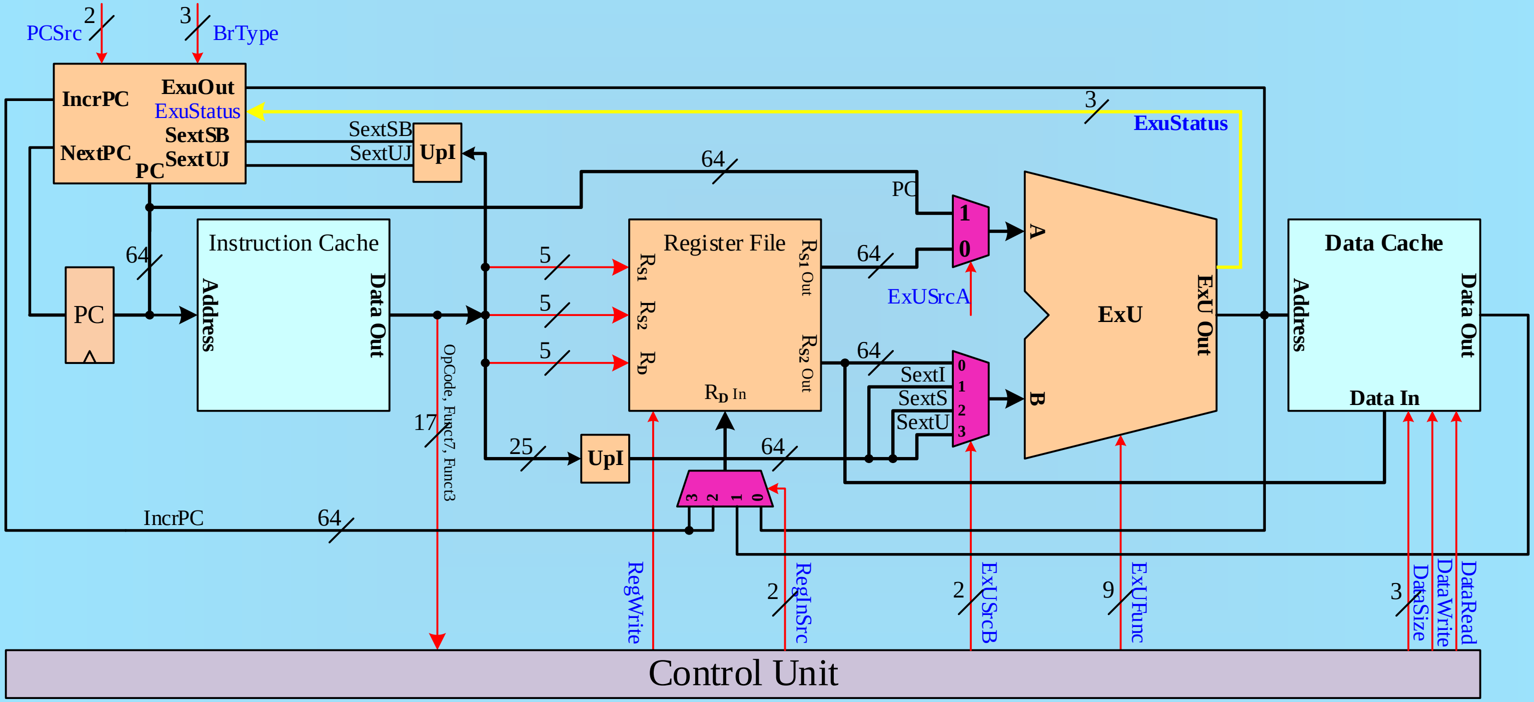
1. The Instruction and Data Caches are also now shared. Instructions are read from the Cache in the first cycle of every instruction and placed in a register called the **Instruction Register**. A load or store instruction may access the cache in any later cycle

2. The **addition** that was performed by the **Next Address Logic** can now be performed by the **ExU**. Every Instruction Cycle begins with the current PC being used as an address to the Cache to Read a word and place it in an **Instruction Register**.

3. -**Holding registers** must be placed at the outputs of specific circuits to ensure that values produced in a given clock cycle are available in subsequent cycles.
   The holding registers *may or may not* require Write Enable signals.

The Single-Cycle Datapath

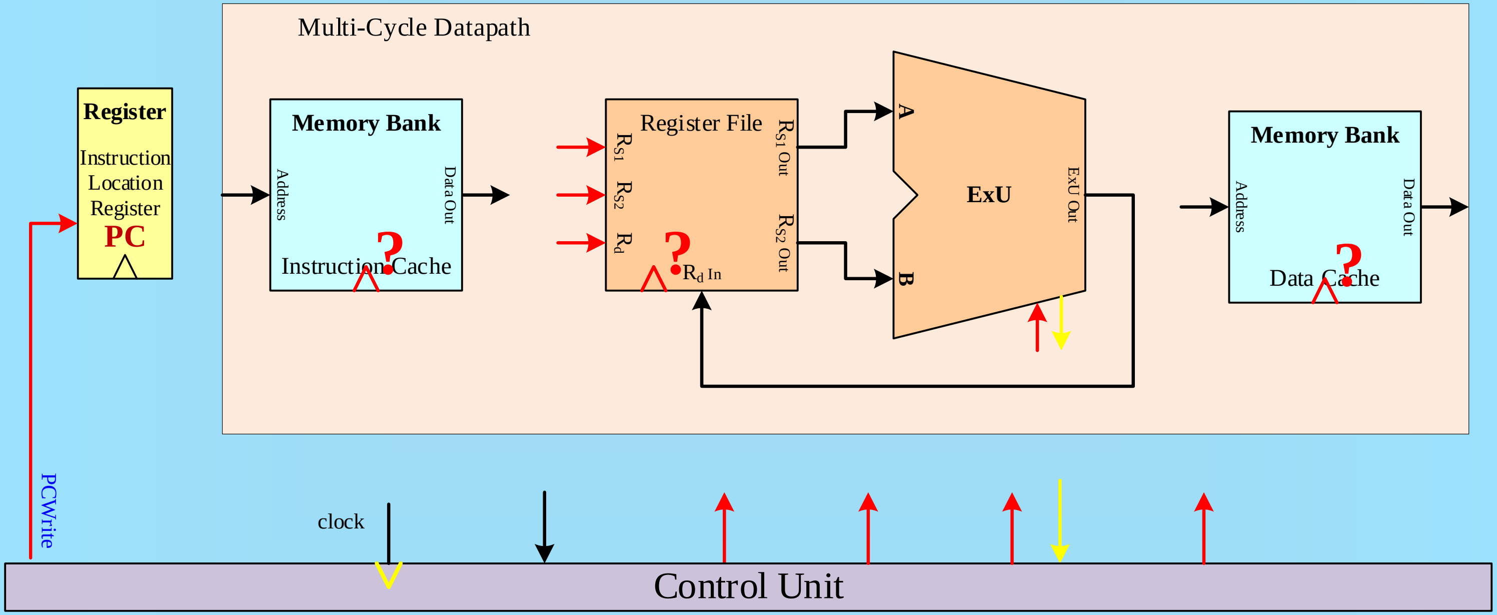
As a reminder, here is the single-cycle datapath from the previous lecture.



Multi-Cycle Datapath – PC and Control Unit

Multi-cycle operation:

- For conceptual simplicity, let's consider the memory caches as part of the datapath.
- The Control Unit Outputs will have different values for each clock cycle thus the control unit must now become a clocked synchronous circuit.
- A write control signal is needed to prevent the PC value from changing on intermediate clock edges, PCWrite.



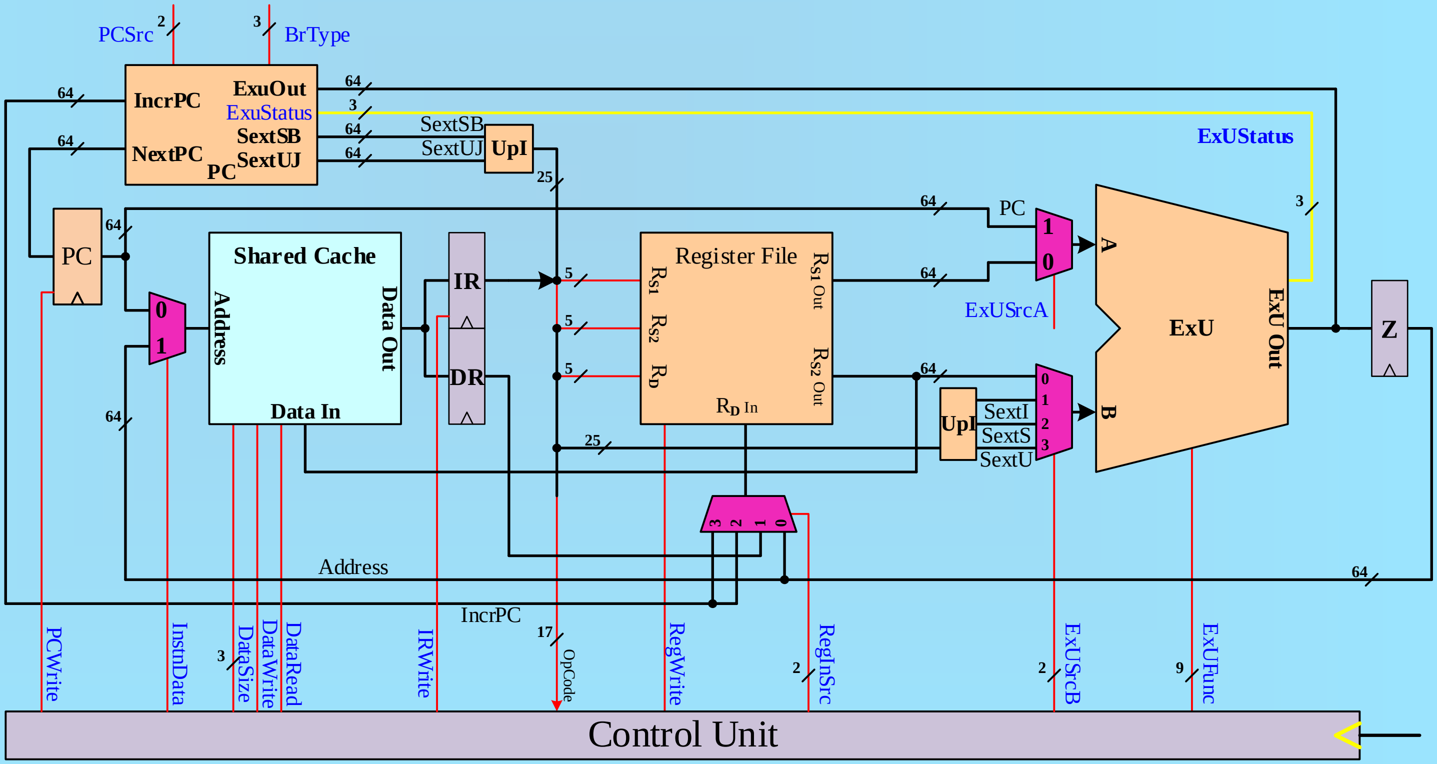
Sharing the Instruction Cache and Data Cache

SFU

-To combine the Instruction cache and the Data Cache a MUX is needed for addressing the memory unit, and two registers, a Data register, **DR**, and Instruction Register, **IR**, to **hold** the value fetched from the memory unit. The Value in the **IR** must NOT be over-written thus a Write-Enable control signal, IRWrite is required. We also use one of the reserved codes for DataSize, for Instruction Read.

-A holding register, **Z**, is inserted at the output of the Execution Unit. (Is this really necessary?) The registers, **Z** and **DR** clock on every cycle. No write enables are necessary because the effect of meaningless values can easily be discarded by controlling RegInSrc/ RegWrite and DataWrite.

DataSize			Write OP	Read Op
0	0	0	byte	Ubyte
0	0	1	half	Uhalf
0	1	0	word	Uword
0	1	1	double	double
1	0	0	rsvd	Sbyte
1	0	1	rsvd	Shalf
1	1	0	rsvd	Sword
1	1	1	rsvd	INST

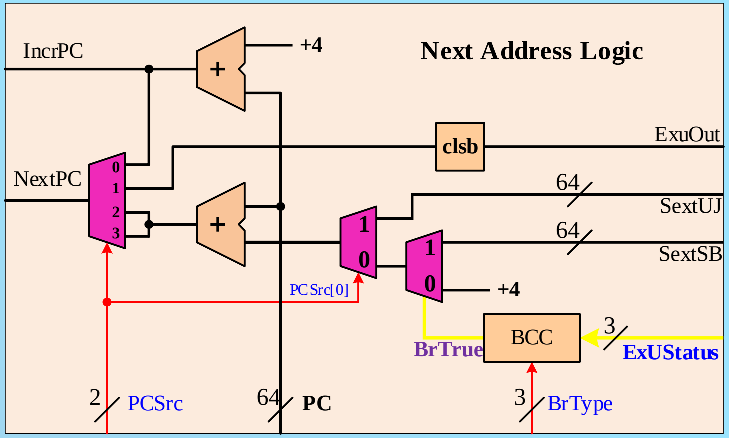


Breaking Apart the Next Address Logic

We may eliminate both adders. The calculations

1. PC ← PC+(2*imm20),
2. PC ← PC+(2*imm12),
3. PC ← PC+4, and
4. R_D ← PC+4

may be calculated by the Execution Unit. A path from the PC to ExU already exist for the **auipc** instruction. The ExUSrcB MUX must be expanded to include the immediate sources **SextSB** and **SextUJ**.



The location of the **BCC** circuit is a minor philosophical point. Let’s place it within the Control Unit to reduce the number of control signals in the diagram. In principle, if ExUStatus are Control Unit Inputs, the control unit can decode the Branch Type and can assess BrTrue. Constants used in the PC address calculation can thus be selected with the ExUSrcB control signals.
We shall define the signals **ExUStatus <= zero & AltB & AltBu**;

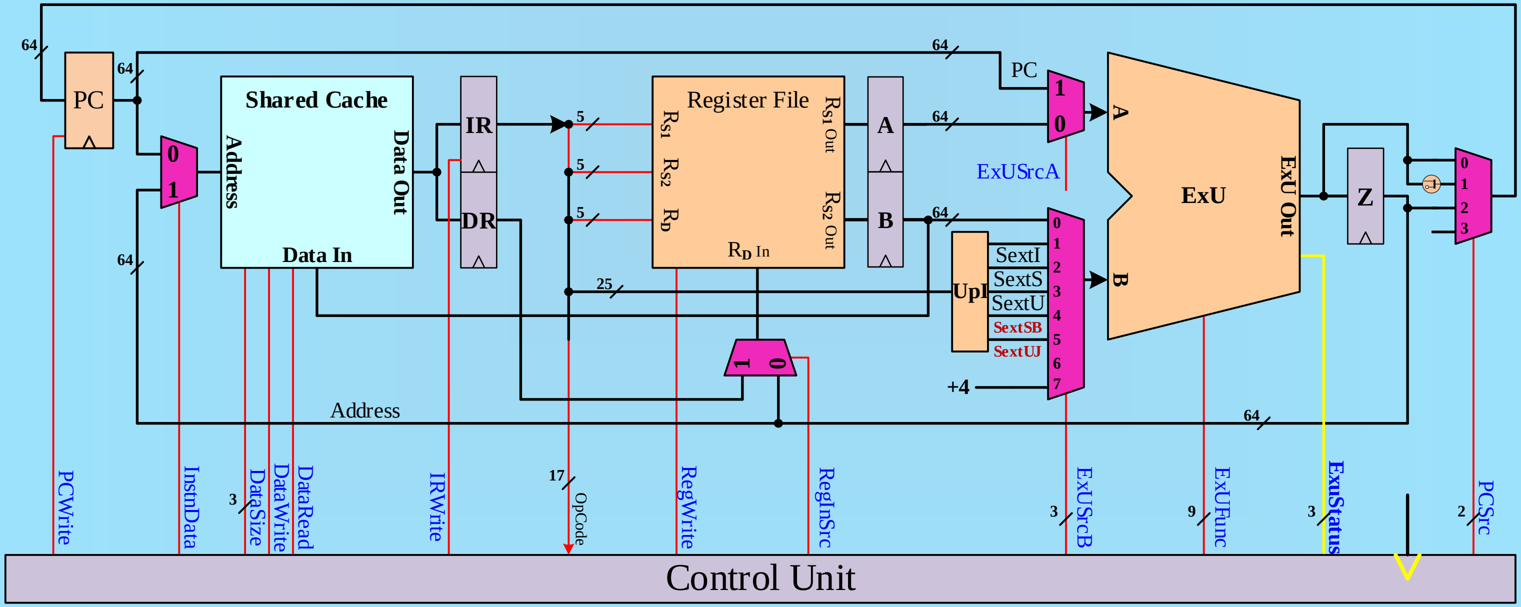
The **NextPC** MUX could be implemented with only 2-channels but it has been left as a 4-channel in anticipation that **ebreak** and **ecall** instructions would use an extra channel.
Clearing the **lsb** is a fictitious action. (How many flip-flops are needed for the PC)
The MUX that selects the source for $R_D$in may also be a 2-channels since **IncrPC** is no longer needed because PC+4 comes from the output of the Execution Unit.

The Multi-Cycle Datapath

Finally, the last modification: two holding registers, **A** and **B**, are inserted between the Register file and the Execution Unit.

These registers allow Register Operand Read to occur in a clock cycle before the primary operation.

These Registers do not need Write Enables since any results appearing at the output of the Execution Unit due to meaningless values in **A** or **B** can easily be discarded by de-asserting RegWrite and/or PCWrite.



Changes to the MUXs

PCSrc		source
0	0	ExUout
0	1	clear lsb
1	0	Z-Reg
1	1	

The PC-Source Multiplexer:

The ExUSrcB - Multiplexer:

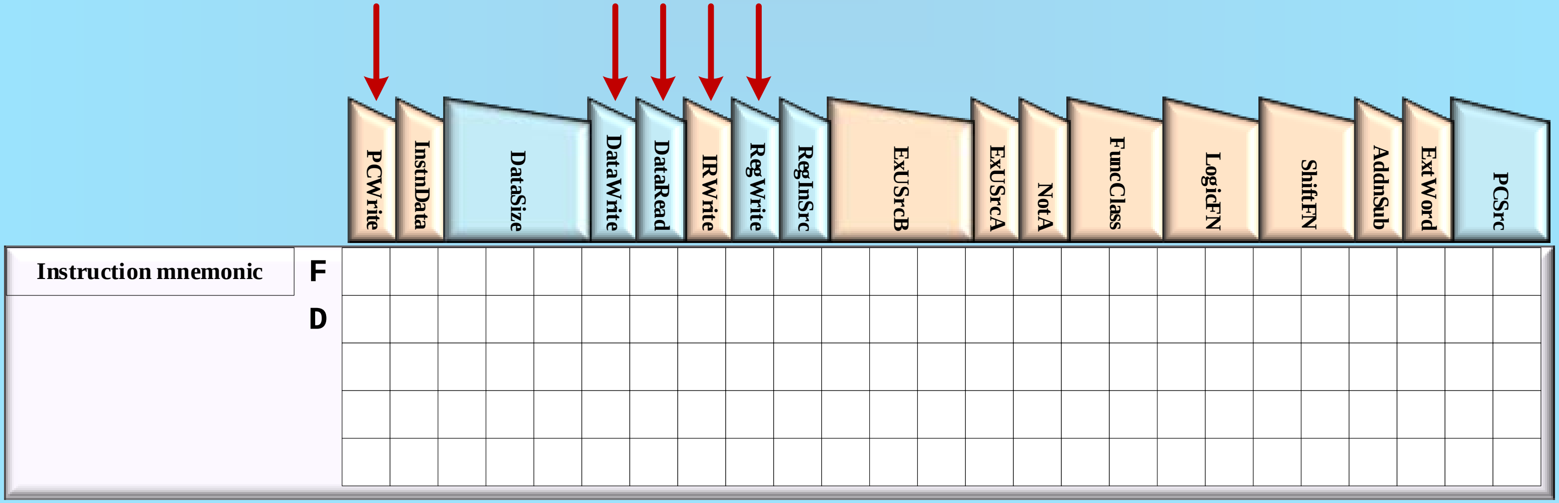
ExUSrcB			Write OP
0	0	0	RegB
0	0	1	SextI
0	1	0	SextS
0	1	1	SextU
1	0	0	SextSB
1	0	1	SextUJ
1	1	0	
1	1	1	+4

The Outputs from the Control Unit: (The Control Word)

The 51 instructions may take 3, 4 or 5 clock cycles to complete.

During each clock cycle, the control word must have appropriate values that determine specific actions for each cycle.

The Write Enable signals must never be “don’t care”. (*and DataRead*)



		PCWrite	InstnData	DataSize			DataWrite	DataRead	IRWrite	RegWrite	RegInSrc	ExUSrcB			ExUSrcA	NotA	FuncClass		LogicFN		ShiftFN		AddnSub	ExtWord	PCSrc	
Instruction mnemonic	**F**																									
	D																									

The 1st Cycle - Instruction Fetch

The First cycle is the same for all instructions. (**Fetch**)

Two actions occur during this cycle.

1) The Machine Instruction is fetched from memory (read) and stored in the Instruction Register.

2) PC + 4 is calculated by the Execution Unit and the result, placed back in the PC.

InstnData, DataSize, DataWrite & DataRead must be controlled appropriately to allow the memory read action. In addition, IRWrite must be asserted during the first cycle to ensure that the Instruction be clocked into the Instruction Register at the end of this cycle.

The ExuSrcA & ExUSrcB MUXes must be selected for **PC** + 4.
Furthermore, the ExUFunc control lines must be chosen to perform an addition operation.
(remember that ShiftFN must be "**00**")

Since the result must be written back into the **PC** at the end of the first cycle, PCWrite must be asserted and also the ExUOut (bypassing the Z-Reg) must be directed to the PC by selecting the PC source, PCSrc = "**00**".

	PCWrite	InstnData	DataSize	DataWrite	DataRead	IRWrite	RegWrite	RegInSrc	ExUSrcB	ExUSrcA	NotA	FuncClass	LogicFN	ShiftFN	AddnSub	ExtWord	PCSrc
F	1	0	1 1 1	0	1	1	0		1 1 1	1	0	1 0		0 0	0	0	0 0

The 2nd Cycle – Instruction Decode

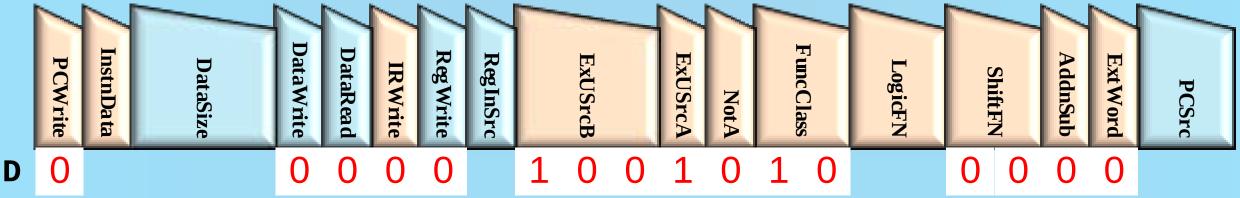
The Second cycle is also the same for all instructions. (**Decode**)

Three actions occur during this cycle.

1) The 17 opcode, func7 & func3 bits, stored in the Instruction Register are applied as inputs to the control unit and the instruction is effectively decoded.

2) The Register Address Fields, R_{S1} and R_{S2}, stored in the Instruction Register, are applied to the register file, and the values are read into the holding registers, A-Reg and B-Reg.

3) The Execution Unit uses the immediate field, **SextSB** and the **PC**, to calculate a Branch Target Address. The result is clocked into the Z-Reg at the end of this cycle.
It is possible that the value is nonsense because the instruction may not be a branch instruction.

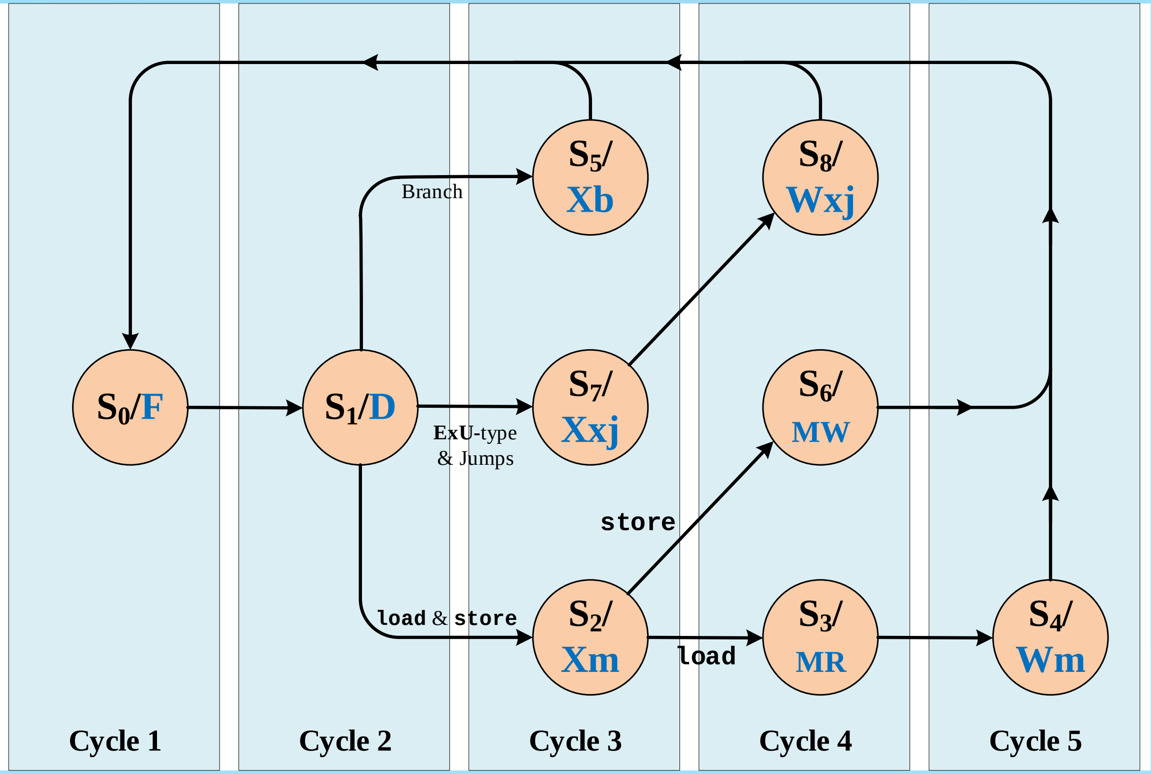


	PCWrite	InstnData	DataSize	DataWrite	DataRead	IRWrite	RegWrite	RegInSrc	ExUSrcB	ExUSrcA	NotA	FuncClass	LogicFN	ShiftFN	AddnSub	ExtWord	PCSrc
D	0			0	0	0	0		1 0 0	1	0	1 0		0 0	0	0	

The Control Unit State Machine

After Fetch and Decode, the particular instruction is known to the control unit and thus subsequent control words will depend on the specific instruction.

- The 30 “Execute” class instructions, “jump” class and memory “store” class instructions all complete in four clock cycles.
- Branch instructions can be completed in three cycles,
- and memory “load” class instructions complete in five cycles.
- We may represent the behaviour of the control unit using a “pseudo” state diagram.



“Execute” & “Memory” Class Instructions

Fetch: **F**

- InstnData = ‘**0**’, DataWrite = ‘**0**’, DataRead = ‘**1**’, DataSize = “**111**” ,
- IRWrite = ‘**1**’,
- ExUSrcA = ‘**1**’, ExUSrcB = “**111**” , ExUFunc = *chosen to add*
- PCSrc = “**00**” , PCWrite = ‘**1**’

Decode and Register Access: **D**

- ExUSrcA = ‘**1**’, ExUSrcB = “**100**”, ExUFunc = *chosen to add*

What is this calculation?

Execution of ExU-type: **Xxj**

- ExUSrcA = ‘**0**’, ExUSrcB = “**000**” or “**001**”, ExUFunc = *chosen appropriately*

Register Writeback of ExU-type: **Wxj**

- RegInSrc = ‘**0**’, RegWrite = ‘**1**’

When did the value get into the z-register?

Memory Address Calculation for Mem-type: **Xm**

Does the B-Reg have the correct value in the 4th cycle?

- ExUSrcA = ‘**0**’, ExUSrcB = “**001**” or “**010**”, ExUFunc = *chosen to add*

Memory Write for **sd, sw, sh, sb** instructions: **MW**

alway’s choose DataSize(2) = ‘**0**’

- InstnData = ‘**1**’, DataWrite = ‘**1**’, DataRead = ‘**0**’, DataSize = “**???**” *chosen appropriately*

Memory Read for **ld, lw, lh, lb, lwu, lhu, lbu** instructions: **MR**

- InstnData = ‘**1**’, DataWrite = ‘**0**’, DataRead = ‘**1**’ , DataSize = “**???**” *chosen appropriately*

Register Write for **load** instructions: **Wm**

- RegInSrc = ‘**1**’, RegWrite = ‘**1**’

When did the value get into the DR-register?

Control Words for Flow Control Instructions

Third Cycle for Branch type instructions: **Xb**

- ExUSrcA = ‘**0**’, ExUSrcB = “**000**”, ExUFunc = *chosen to subtract* - ***do not choose*** *slt or sltu*

  also choose ShiftFN = “**00**”
- PCSrc = “**10**”, since the BTA is in the Z-Reg.
- PCWrite = ‘**0**’ or ‘**1**’. Conditional on **AltB**, **AltBu** and **Zero**.

Formally some of the outputs are conditional outputs, so the state diagram is not technically correct.

Jump and link, **`jal`**, & jump and link Register, **`jalr`**, require two additions.

PC + 4 is recomputed in the 3rd cycle and the value transferred to the Z-Reg.

Two actions occur during the 4th cycle,

1. Register Writeback occurs storing the Z-Reg into the register file.
2. The Jump Target Address is calculated and transferred to the **PC** in the same cycle.

Third Cycle for Jump type instructions: **Xxj**

- ExUSrcA = ‘**1**’, ExUSrcB = “**111**” , ExUFunc = *chosen to add*

Fourth Cycle for Jump type instructions: **Wxj**

- ExUSrcA = ‘**1**’, ExUSrcB = “**101**” , ExUFunc = *chosen to add (for* ***`jal`****)*

  or ExUSrcA = ‘**0**’, ExUSrcB = “**001**” , ExUFunc = *chosen to add (for* ***`jalr`****)*
- PCSrc = “**00**”, for ***`jal`*** , PCSrc = “**01**”, for ***`jalr`***, *to ensure clearing the lsb*.
- RegInSrc = ‘**0**’, RegWrite = ‘**1**’ and PCWrite = ‘**1**’

Analysis of Multi-Cycle Performance

The Execution Unit is the bottleneck element.
The Clock period cannot be shorter than 4 ns. (250 MHz)

Suppose that 100 instructions execute.

ExU-type	**44*4 = 176 cycles**
Store-type	**12*4 = 48 cycles**
Load-type	**24*5 = 120 cycles**
Branch-type	**18*3 = 54 cycles**
Jump-type	**2*4 = 8 cycles**

Total = 406 cycles => 1624 ns
Thus 100 instructions takes 1624 ns
Ideally, we have on average
16.24 ns per instruction.
or the throughput (reciprocal) is
61.57 MIPS

Notice that the performance is worse than the Single-Cycle Processor (**76.92 MIPS**)
Suppose the Execution Unit is broken into two elements, each with a delay of 2 ns. Can the performance be improved? Calculate this yourself.