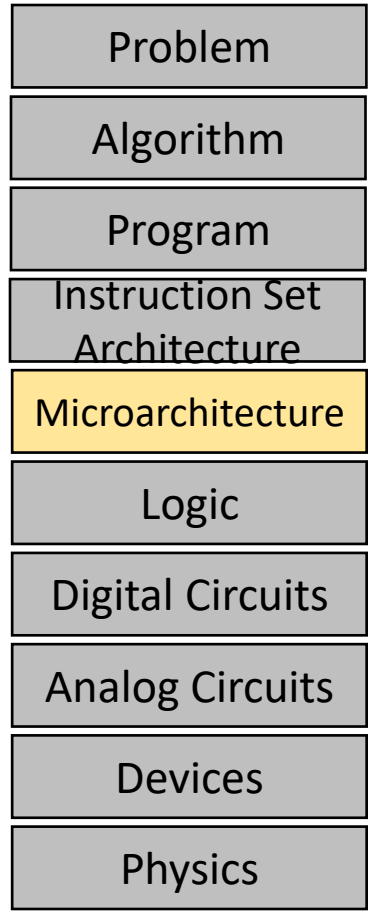
Universität Bremen

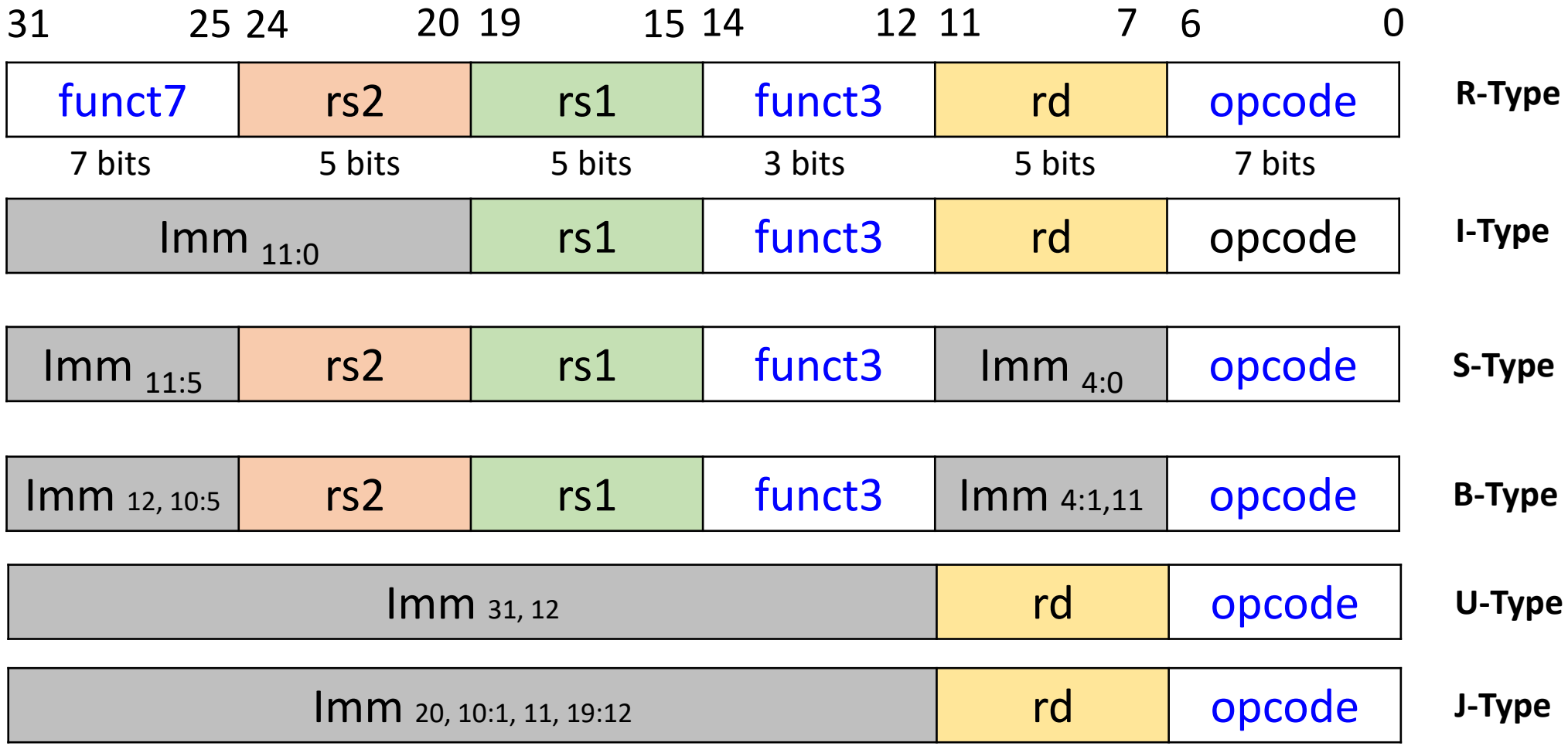
RISC-V Processor Architecture

Prof. Dr. Rolf Drechsler
Dr. Muhammad Hassan
M.Sc. Jan Zielasko
M.Sc. Milan Funck



Universität Bremen

Discussed the last time …



Universität Bremen

Today's Agenda

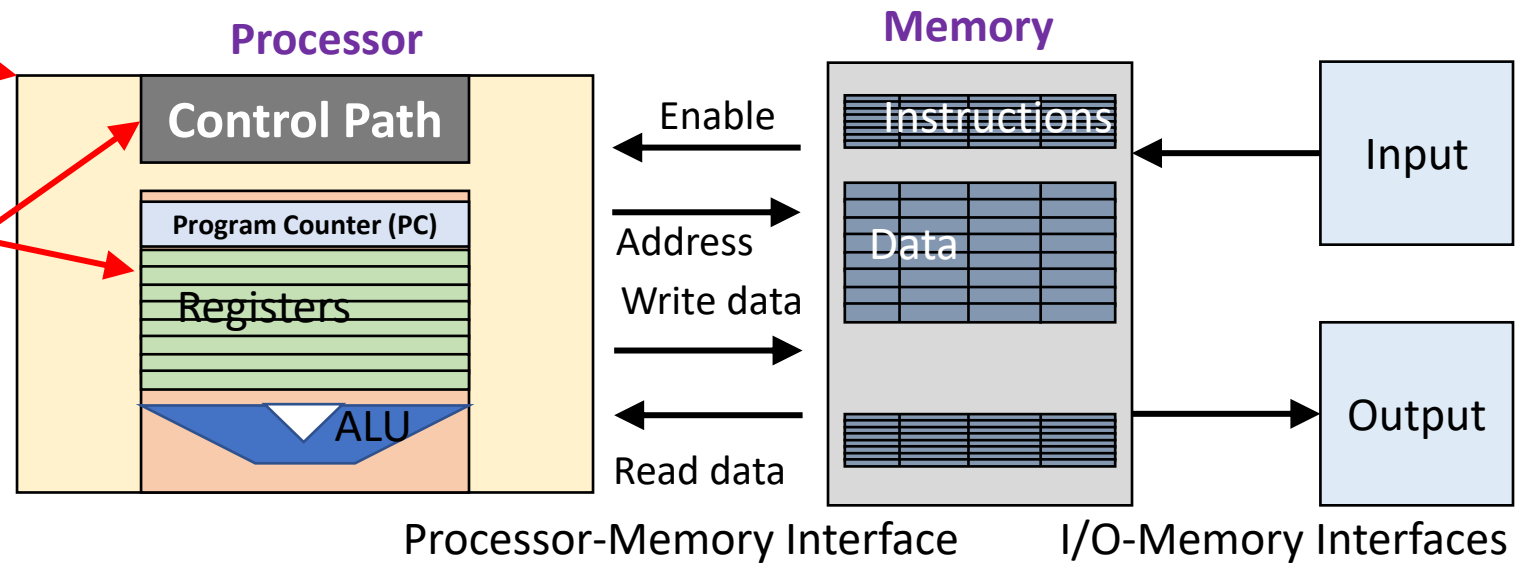
- An Overview of RISC-V Implementation
  - Datapath
  - Controller
- Building blocks
  - Program Counter (PC)
  - Arithmetic Logic Unit (ALU)
  - Instruction Memory
  - Data Memory
  - Registers
- Implementing ADD/SUB
- Adding ADDI Instruction
  - Immediate generator

Universität Bremen

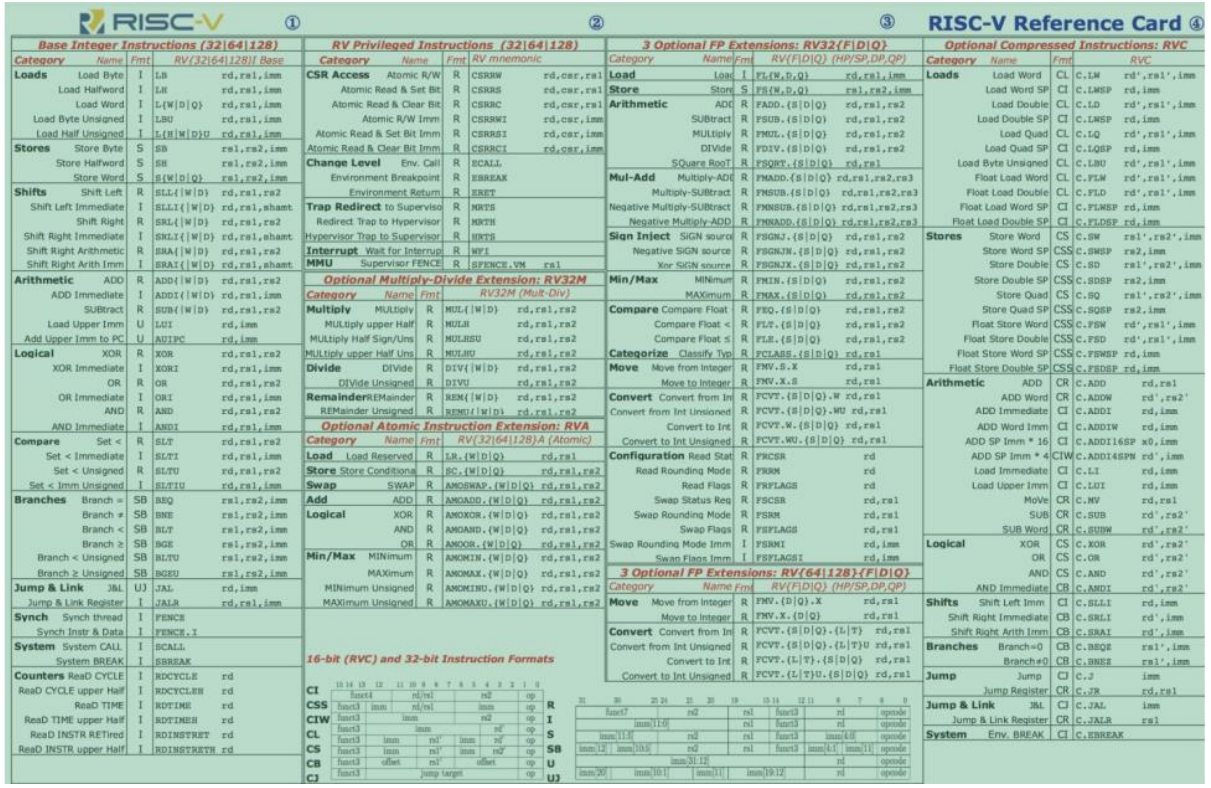
IPhone Launch

How Do We Build a Single Cycle Processor?

- Processor (CPU)
  - The active part of the computer that does all the work (data manipulation and decision making)
- Datapath (“the brawn”)
  - portion of the processor that contains hardware necessary to perform operations required by the processor.
- Control (“the brain”)
  - portion of the processor (also in hardware) that tells the datapath what needs to be done.



Goal – Design HW to Execute All RV32I Instructions


RISC-V ① ② ③ RISC-V Reference Card ④

Base Integer Instructions (32|64|128)

Category	Name	Fmt	RV{32|64|128}I Base
Loads	Load Byte	I	LB rd,rs1,imm
	Load Halfword	I	LH rd,rs1,imm
	Load Word	I	L{W|D|Q} rd,rs1,imm
	Load Byte Unsigned	I	LBU rd,rs1,imm
	Load Half Unsigned	I	L{H|W|D}U rd,rs1,imm
Stores	Store Byte	S	SB rs1,rs2,imm
	Store Halfword	S	SH rs1,rs2,imm
	Store Word	S	S{W|D|Q} rs1,rs2,imm
Shifts	Shift Left	R	SLL{|W|D} rd,rs1,rs2
	Shift Left Immediate	I	SLLI{|W|D} rd,rs1,shamt
	Shift Right	R	SRL{|W|D} rd,rs1,rs2
	Shift Right Immediate	I	SRLI{|W|D} rd,rs1,shamt
	Shift Right Arithmetic	R	SRA{|W|D} rd,rs1,rs2
	Shift Right Arith Imm	I	SRAI{|W|D} rd,rs1,shamt
Arithmetic	ADD	R	ADD{|W|D} rd,rs1,rs2
	ADD Immediate	I	ADDI{|W|D} rd,rs1,imm
	SUBtract	R	SUB{|W|D} rd,rs1,rs2
	Load Upper Imm	U	LUI rd,imm
	Add Upper Imm to PC	U	AUIPC rd,imm
Logical	XOR	R	XOR rd,rs1,rs2
	XOR Immediate	I	XORI rd,rs1,imm
	OR	R	OR rd,rs1,rs2
	OR Immediate	I	ORI rd,rs1,imm
	AND	R	AND rd,rs1,rs2
	AND Immediate	I	ANDI rd,rs1,imm
Compare	Set <	R	SLT rd,rs1,rs2
	Set < Immediate	I	SLTI rd,rs1,imm
	Set < Unsigned	R	SLTU rd,rs1,rs2
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm
Branches	Branch =	SB	BEQ rs1,rs2,imm
	Branch ≠	SB	BNE rs1,rs2,imm
	Branch <	SB	BLT rs1,rs2,imm
	Branch ≥	SB	BGE rs1,rs2,imm
	Branch < Unsigned	SB	BLTU rs1,rs2,imm
	Branch ≥ Unsigned	SB	BGEU rs1,rs2,imm
Jump & Link	J&L	UJ	JAL rd,imm
	Jump & Link Register	I	JALR rd,rs1,imm
Synch	Synch thread	I	FENCE
	Synch Instr & Data	I	FENCE.I
System	System CALL	I	SCALL
	System BREAK	I	SBREAK
Counters	ReaD CYCLE	I	RDCYCLE rd
	ReaD CYCLE upper Half	I	RDCYCLEH rd
	ReaD TIME	I	RDTIME rd
	ReaD TIME upper Half	I	RDTIMEH rd
	ReaD INSTR RETired	I	RDINSTRET rd
	ReaD INSTR upper Half	I	RDINSTRETH rd

RV Privileged Instructions (32|64|128)

Category	Name	Fmt	RV mnemonic
CSR Access	Atomic R/W	R	CSRRW rd,csr,rs1
	Atomic Read & Set Bit	R	CSRRS rd,csr,rs1
	Atomic Read & Clear Bit	R	CSRRC rd,csr,rs1
	Atomic R/W Imm	R	CSRRWI rd,csr,imm
	Atomic Read & Set Bit Imm	R	CSRRSI rd,csr,imm
	Atomic Read & Clear Bit Imm	R	CSRRCI rd,csr,imm
Change Level	Env. Call	R	ECALL
	Environment Breakpoint	R	EBREAK
	Environment Return	R	ERET
Trap Redirect	to Supervisor	R	MRTS
	Redirect Trap to Hypervisor	R	MRTH
	Hypervisor Trap to Supervisor	R	HRTS
Interrupt	Wait for Interrup	R	WFI
MMU	Supervisor FENCE	R	SFENCE.VM rs1

Optional Multiply-Divide Extension: RV32M

Category	Name	Fmt	RV32M (Mult-Div)
Multiply	MULtiply	R	MUL{|W|D} rd,rs1,rs2
	MULtiply upper Half	R	MULH rd,rs1,rs2
	MULtiply Half Sign/Uns	R	MULHSU rd,rs1,rs2
	MULtiply upper Half Uns	R	MULHU rd,rs1,rs2
Divide	DIVide	R	DIV{|W|D} rd,rs1,rs2
	DIVide Unsigned	R	DIVU rd,rs1,rs2
Remainder	REMainder	R	REM{|W|D} rd,rs1,rs2
	REMainder Unsigned	R	REMU{|W|D} rd,rs1,rs2

Optional Atomic Instruction Extension: RVA

Category	Name	Fmt	RV{32|64|128}A (Atomic)
Load	Load Reserved	R	LR.{W|D|Q} rd,rs1
Store	Store Conditiona	R	SC.{W|D|Q} rd,rs1,rs2
Swap	SWAP	R	AMOSWAP.{W|D|Q} rd,rs1,rs2
Add	ADD	R	AMOADD.{W|D|Q} rd,rs1,rs2
Logical	XOR	R	AMOXOR.{W|D|Q} rd,rs1,rs2
	AND	R	AMOAND.{W|D|Q} rd,rs1,rs2
	OR	R	AMOOR.{W|D|Q} rd,rs1,rs2
Min/Max	MINimum	R	AMOMIN.{W|D|Q} rd,rs1,rs2
	MAXimum	R	AMOMAX.{W|D|Q} rd,rs1,rs2
	MINimum Unsigned	R	AMOMINU.{W|D|Q} rd,rs1,rs2
	MAXimum Unsigned	R	AMOMAXU.{W|D|Q} rd,rs1,rs2

16-bit (RVC) and 32-bit Instruction Formats

3 Optional FP Extensions: RV32{F|D|Q}

Category	Name	Fmt	RV{F|D|Q} (HP/SP,DP,QP)
Load	Load	I	FL{W,D,Q} rd,rs1,imm
Store	Store	S	FS{W,D,Q} rs1,rs2,imm
Arithmetic	ADD	R	FADD.{S|D|Q} rd,rs1,rs2
	SUBtract	R	FSUB.{S|D|Q} rd,rs1,rs2
	MULtiply	R	FMUL.{S|D|Q} rd,rs1,rs2
	DIVide	R	FDIV.{S|D|Q} rd,rs1,rs2
	SQuare RooT	R	FSQRT.{S|D|Q} rd,rs1
Mul-Add	Multiply-ADd	R	FMADD.{S|D|Q} rd,rs1,rs2,rs3
	Multiply-SUBtract	R	FMSUB.{S|D|Q} rd,rs1,rs2,rs3
	Negative Multiply-SUBtract	R	FNMSUB.{S|D|Q} rd,rs1,rs2,rs3
	Negative Multiply-ADD	R	FNMADD.{S|D|Q} rd,rs1,rs2,rs3
Sign Inject	SIGN source	R	FSGNJ.{S|D|Q} rd,rs1,rs2
	Negative SIGN source	R	FSGNJN.{S|D|Q} rd,rs1,rs2
	Xor SIGN source	R	FSGNJX.{S|D|Q} rd,rs1,rs2
Min/Max	MINimum	R	FMIN.{S|D|Q} rd,rs1,rs2
	MAXimum	R	FMAX.{S|D|Q} rd,rs1,rs2
Compare	Compare Float =	R	FEQ.{S|D|Q} rd,rs1,rs2
	Compare Float <	R	FLT.{S|D|Q} rd,rs1,rs2
	Compare Float ≤	R	FLE.{S|D|Q} rd,rs1,rs2
Categorize	Classify Typ	R	FCLASS.{S|D|Q} rd,rs1
Move	Move from Integer	R	FMV.S.X rd,rs1
	Move to Integer	R	FMV.X.S rd,rs1
Convert	Convert from Int	R	FCVT.{S|D|Q}.W rd,rs1
	Convert from Int Unsigned	R	FCVT.{S|D|Q}.WU rd,rs1
	Convert to Int	R	FCVT.W.{S|D|Q} rd,rs1
	Convert to Int Unsigned	R	FCVT.WU.{S|D|Q} rd,rs1
Configuration	Read Stat	R	FRCSR rd
	Read Rounding Mode	R	FRRM rd
	Read Flags	R	FRFLAGS rd
	Swap Status Reg	R	FSCSR rd,rs1
	Swap Rounding Mode	R	FSRM rd,rs1
	Swap Flags	R	FSFLAGS rd,rs1
	Swap Rounding Mode Imm	I	FSRMI rd,imm
	Swap Flags Imm	I	FSFLAGSI rd,imm

3 Optional FP Extensions: RV{64|128}{F|D|Q}

Category	Name	Fmt	RV{F|D|Q} (HP/SP,DP,QP)
Move	Move from Integer	R	FMV.{D|Q}.X rd,rs1
	Move to Integer	R	FMV.X.{D|Q} rd,rs1
Convert	Convert from Int	R	FCVT.{S|D|Q}.{L|T} rd,rs1
	Convert from Int Unsigned	R	FCVT.{S|D|Q}.{L|T}U rd,rs1
	Convert to Int	R	FCVT.{L|T}.{S|D|Q} rd,rs1
	Convert to Int Unsigned	R	FCVT.{L|T}U.{S|D|Q} rd,rs1

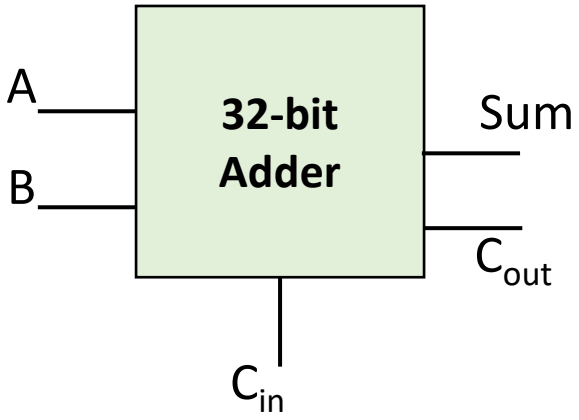
Optional Compressed Instructions: RVC

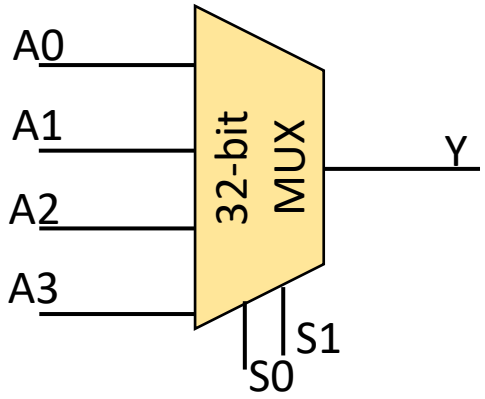
Category	Name	Fmt	RVC
Loads	Load Word	CL	C.LW rd',rs1',imm
	Load Word SP	CI	C.LWSP rd,imm
	Load Double	CL	C.LD rd',rs1',imm
	Load Double SP	CI	C.LWSP rd,imm
	Load Quad	CL	C.LQ rd',rs1',imm
	Load Quad SP	CI	C.LQSP rd,imm
	Load Byte Unsigned	CL	C.LBU rd',rs1',imm
	Float Load Word	CL	C.FLW rd',rs1',imm
	Float Load Double	CL	C.FLD rd',rs1',imm
	Float Load Word SP	CI	C.FLWSP rd,imm
	Float Load Double SP	CI	C.FLDSP rd,imm
Stores	Store Word	CS	C.SW rs1',rs2',imm
	Store Word SP	CSS	C.SWSP rs2,imm
	Store Double	CS	C.SD rs1',rs2',imm
	Store Double SP	CSS	C.SDSP rs2,imm
	Store Quad	CS	C.SQ rs1',rs2',imm
	Store Quad SP	CSS	C.SQSP rs2,imm
	Float Store Word	CSS	C.FSW rd',rs1',imm
	Float Store Double	CSS	C.FSD rd',rs1',imm
	Float Store Word SP	CSS	C.FSWSP rd,imm
	Float Store Double SP	CSS	C.FSDSP rd,imm
Arithmetic	ADD	CR	C.ADD rd,rs1
	ADD Word	CR	C.ADDW rd',rs2'
	ADD Immediate	CI	C.ADDI rd,imm
	ADD Word Imm	CI	C.ADDIW rd,imm
	ADD SP Imm * 16	CI	C.ADDI16SP x0,imm
	ADD SP Imm * 4	CIW	C.ADDI4SPN rd',imm
	Load Immediate	CI	C.LI rd,imm
	Load Upper Imm	CI	C.LUI rd,imm
	MoVe	CR	C.MV rd,rs1
	SUB	CR	C.SUB rd',rs2'
	SUB Word	CR	C.SUBW rd',rs2'
Logical	XOR	CS	C.XOR rd',rs2'
	OR	CS	C.OR rd',rs2'
	AND	CS	C.AND rd',rs2'
	AND Immediate	CB	C.ANDI rd',rs2'
Shifts	Shift Left Imm	CI	C.SLLI rd,imm
	Shift Right Immediate	CB	C.SRLI rd',imm
	Shift Right Arith Imm	CB	C.SRAI rd',imm
Branches	Branch=0	CB	C.BEQZ rs1',imm
	Branch≠0	CB	C.BNEZ rs1',imm
Jump	Jump	CJ	C.J imm
	Jump Register	CR	C.JR rd,rs1
Jump & Link	J&L	CJ	C.JAL imm
	Jump & Link Register	CR	C.JALR rs1
System	Env. BREAK	CI	C.EBREAK

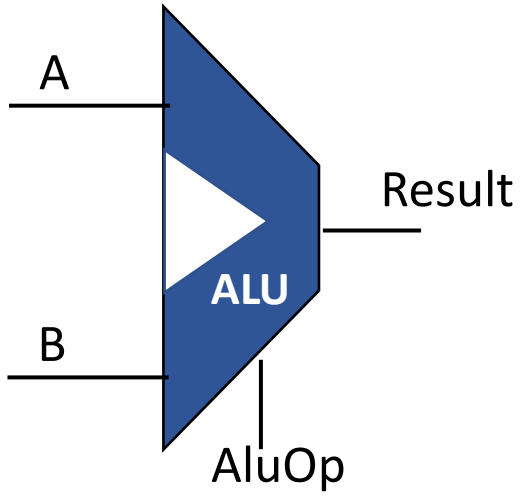
https://www.sifive.com/boards/hifive-unmatched

https://cdn2.hubspot.net/hubfs/3020607/An%20Introduction%20to%20the%20RISC-V%20Architecture.pdf

Combinational Logic Blocks

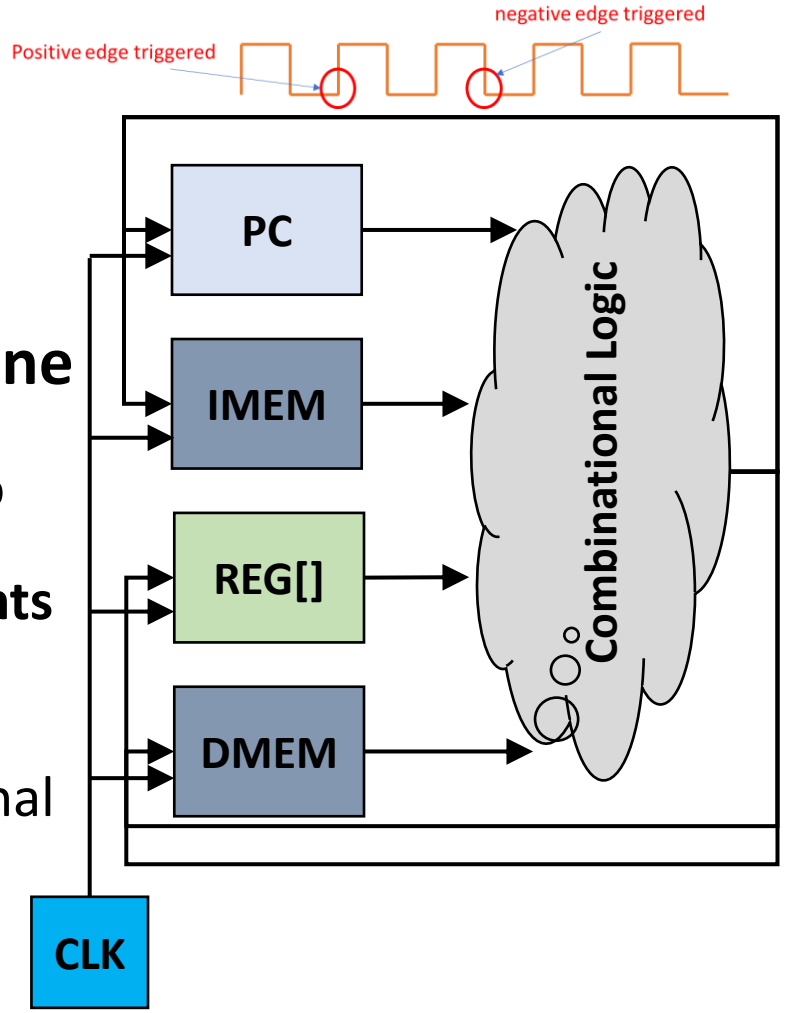




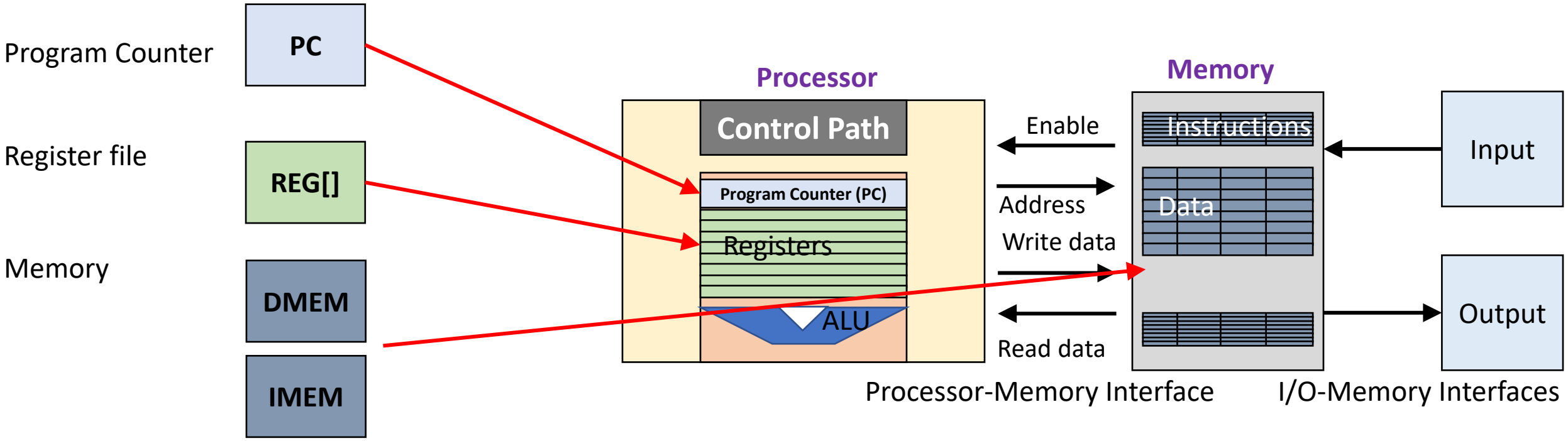


One-Instruction-Per-Cycle RISC-V Machine

- CPU is composed of two types of subcircuits
  - Combination logic blocks
  - State elements
- On every tick of the clock, the computer executes **one instruction**
  - Current outputs of the state elements drive the inputs to **combinational logic**
  - …whose outputs settle at the inputs to the **state elements** before the next rising clock edge.
- At the **rising edge of clock**
  - All the state elements are updated with the combinational logic outputs…
  - and execution moves to the next clock cycle.



State Elements Required by RS32I ISA



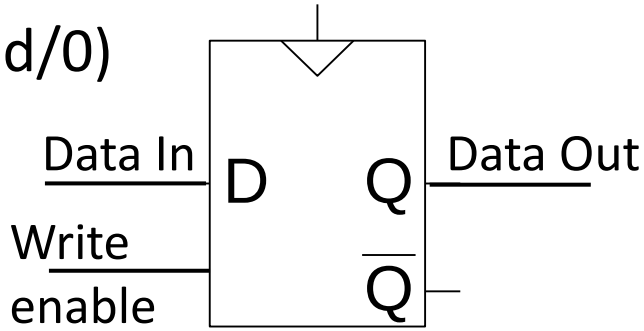
During CPU execution, each RV32I instruction reads and/or updates these state elements.

Program Counter

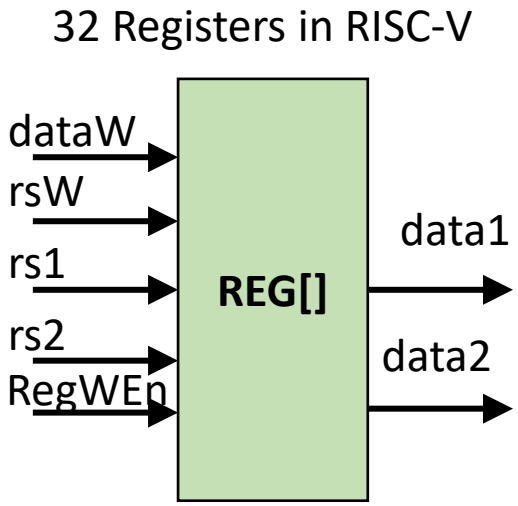
- The **Program Counter** is a 32-bit register

Program Counter

- Input
  - N-bit data input bus
  - Write Enable “Control” bit (1: asserted/high, 0: deasserted/0)
- Output:
  - N-bit data output bus
- Behavior:
  - If Write Enable is 1 on **rising clock edge**, set Data Out=Data In.
  - At all other times, Data Out will not change; it will output its current value.



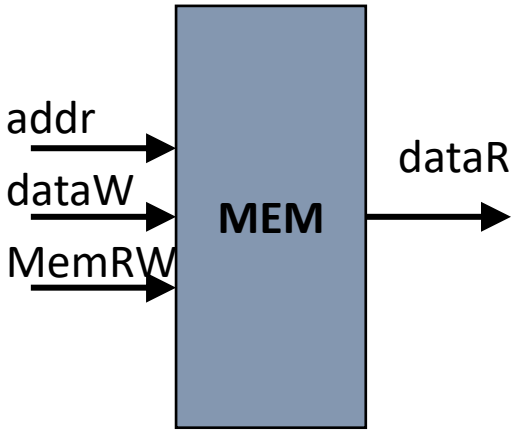
Register File



- Input
  - One 32-bit input data bus, dataW.
  - Three 5-bit select busses, rs1, rs2, and rsW.
  - **RegWEn** control bit.
- Output
  - Two 32-bit output data busses, data1 and data2
- Registers are accessed via their 5-bit register numbers:
  - rs1 selects register to put on data1 bus out.
  - rs2 selects register to put on data2 bus out.
  - rsW selects register to be written via dataW when RegWEn=1.
- Clock behavior: Write operation occurs on rising clock edge.
  - Clock input only a factor on write!
  - All read operations behave like a combinational block:
    - If rs1, rs2 valid, then data1, data2 valid after access time.

Memory

- 32-bit byte-addressed memory space; and
- Memory access with 32-bit words.
- Memory words are accessed as follows:
  - Read: Address addr selects word to put on dataR bus.
  - Write: Set MemRW=1.
  - Address addr selects word to be written with dataW bus.

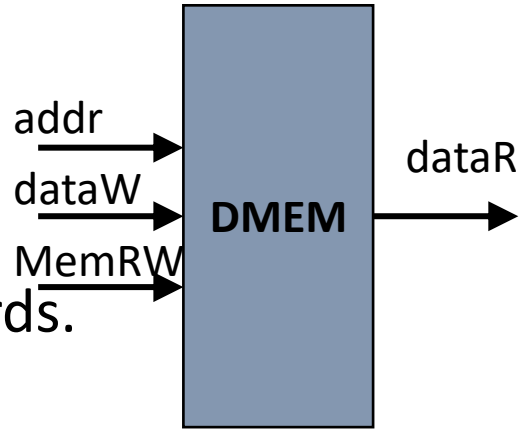


If MemRW=0, MEM behaves like a combinational block.

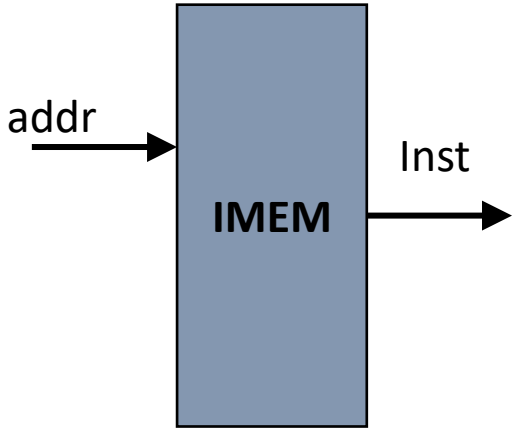
- Like RegFile, clock input is only a factor on write.
  - If MemRW=1, write occurs on rising clock edge.
  - If MemRW=0 and addr valid, then dataR valid after access time.

Two Memories – IMEM and DMEM

- Current abstraction: Memory holds both instructions and data in one contiguous 32-bit memory space.
- In our processor, we'll use two "separate" memories:
  - IMEM: A read-only memory for fetching instructions.
  - DMEM: A memory for loading (read) and storing (write) data words.
  - Under the hood, these are placeholders for caches. (more later)

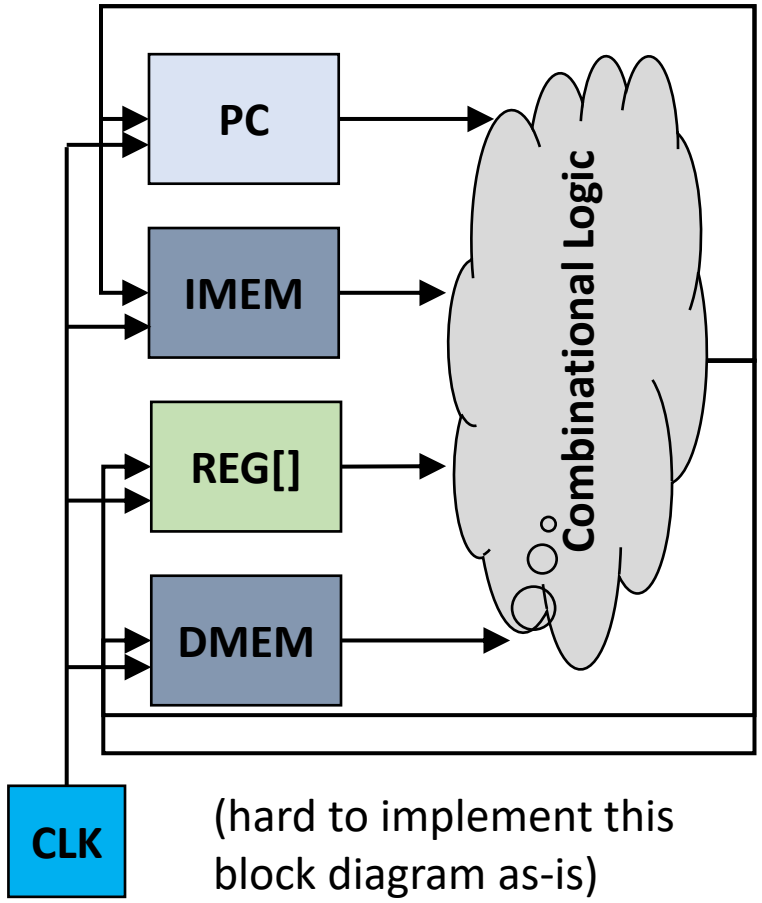


- Because IMEM is read-only, it always behaves like a combinational block:
  - If addr valid, then instr valid after access time.



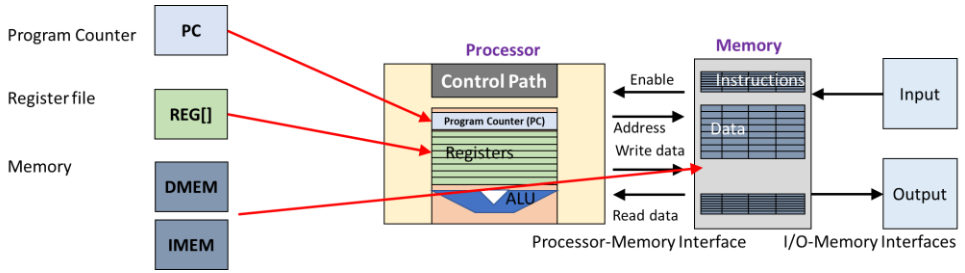
Design the Datapath in Phases

- Task – Execute an instruction
  - All necessary operations, starting with fetching the instruction.
- Problem – A single **monolithic** block would be bulky and inefficient
- Solution – Break up the process into **stages,** then connect the stages to create the whole datapath
  - Smaller stages are easier to design!
  - **Modularity**: Easy to optimize one stage without touching the others.



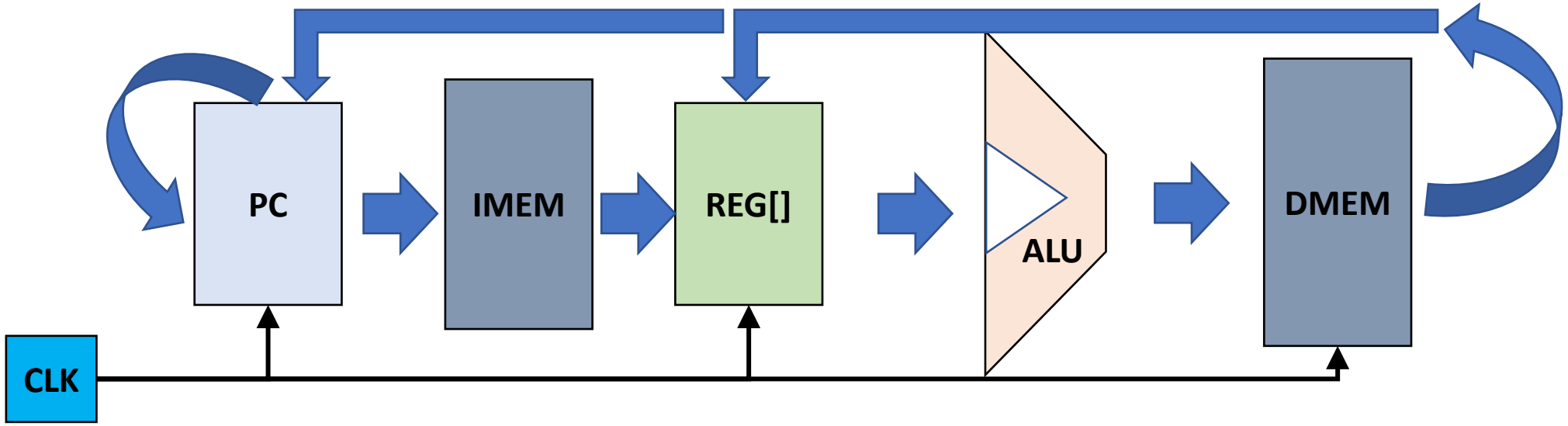
(hard to implement this block diagram as-is)

Stages of Instruction Execution



- 5 basic stages

1. Instruction Fetch (IF)
2. Instruction Decode (ID) + Read Registers
3. Execute (EX) Arithmetic Logic Unit (ALU)
4. Memory Access (MEM)
5. Write back to Register (WB)

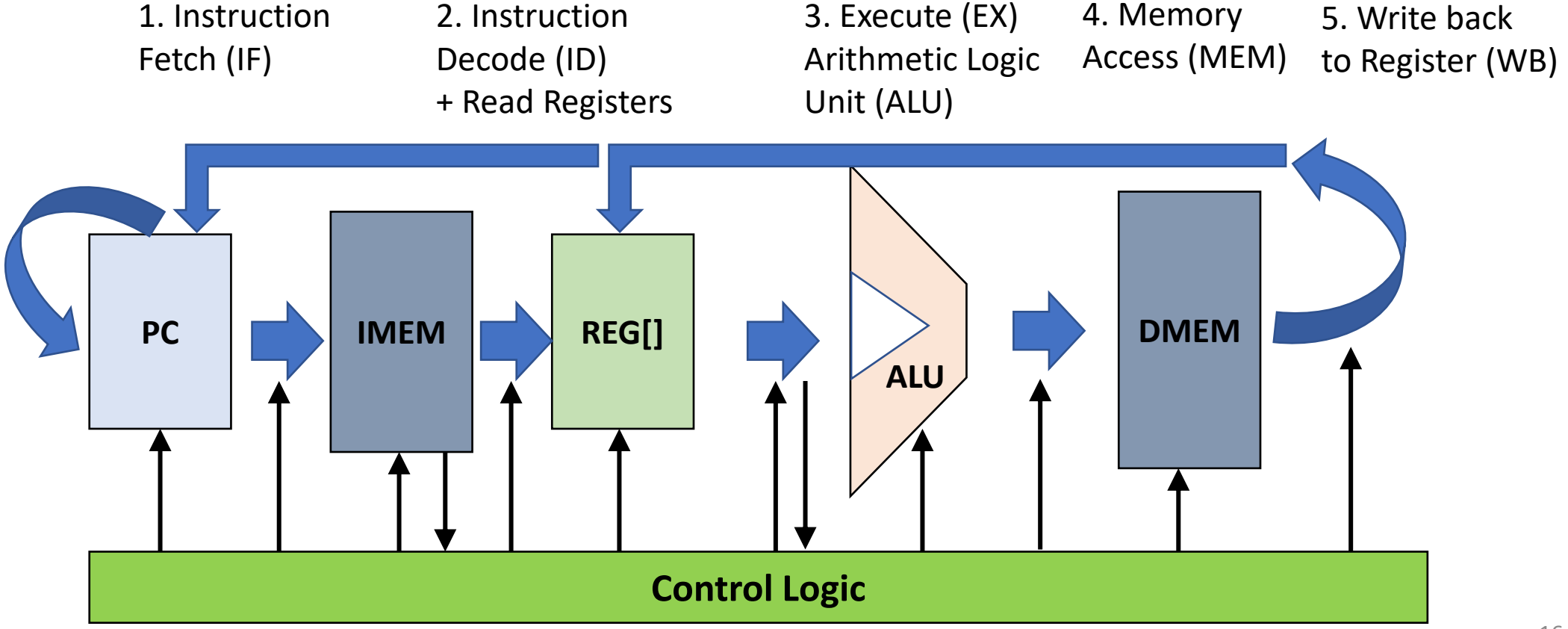


We will implement a single-cycle processor

All stages of one RV32I instruction execute within the same clock cycle.

Not All Instructions Need 5 Stages

- The control logic selects **needed** datapath lines based on the instruction
  - MUX selector, ALU op selector, write enable, etc.

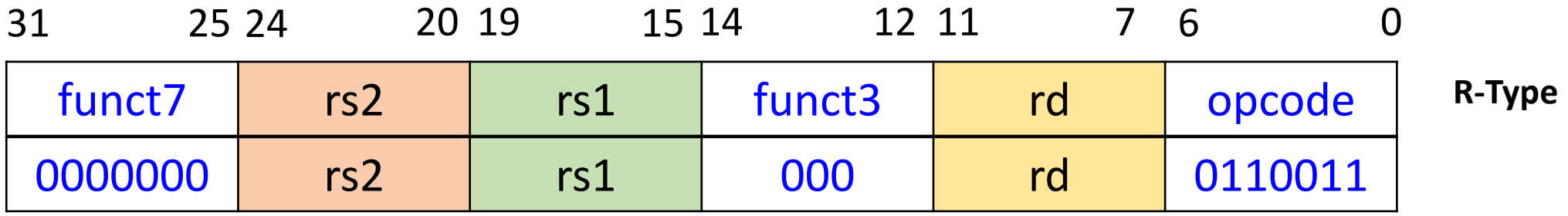


Implementing the add Instruction

Example add-only program

```
0x100 add x18,x18,x10
0x108 add x18,x18,x18
```

- Suppose we had a single instruction in our RISC-V ISA
  - add **add rd, rs1, rs2**



The **add** instructions makes **two changes** to the processor state

- RegFile Reg[rd] = Reg[rs1] + Reg[rs2]
- PC PC = PC + 4

Datapath for add

PC = PC + 4

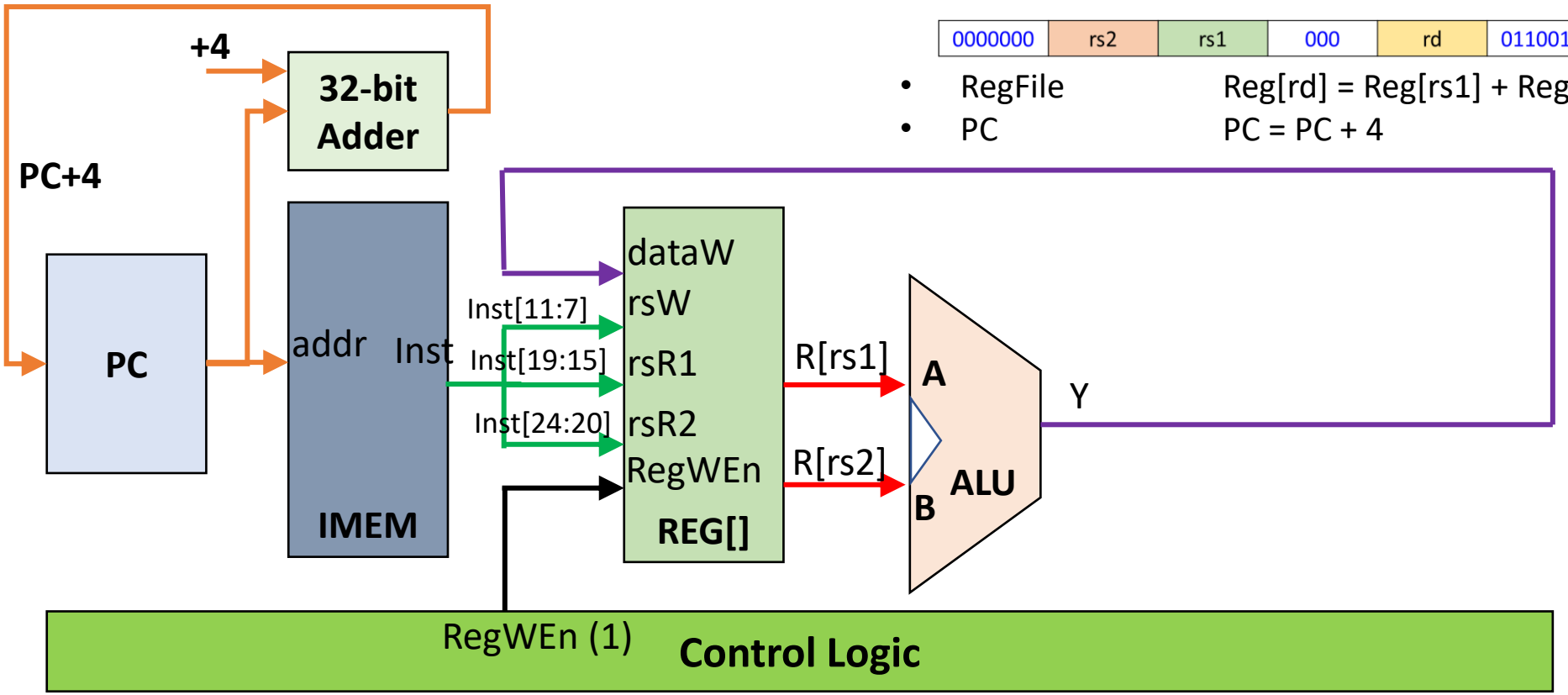
Increment PC to next instruction.

Split instruction to index into RegFile.

R[rd] = R[rs1] + R[rs2]

Feed read register values into ALU.

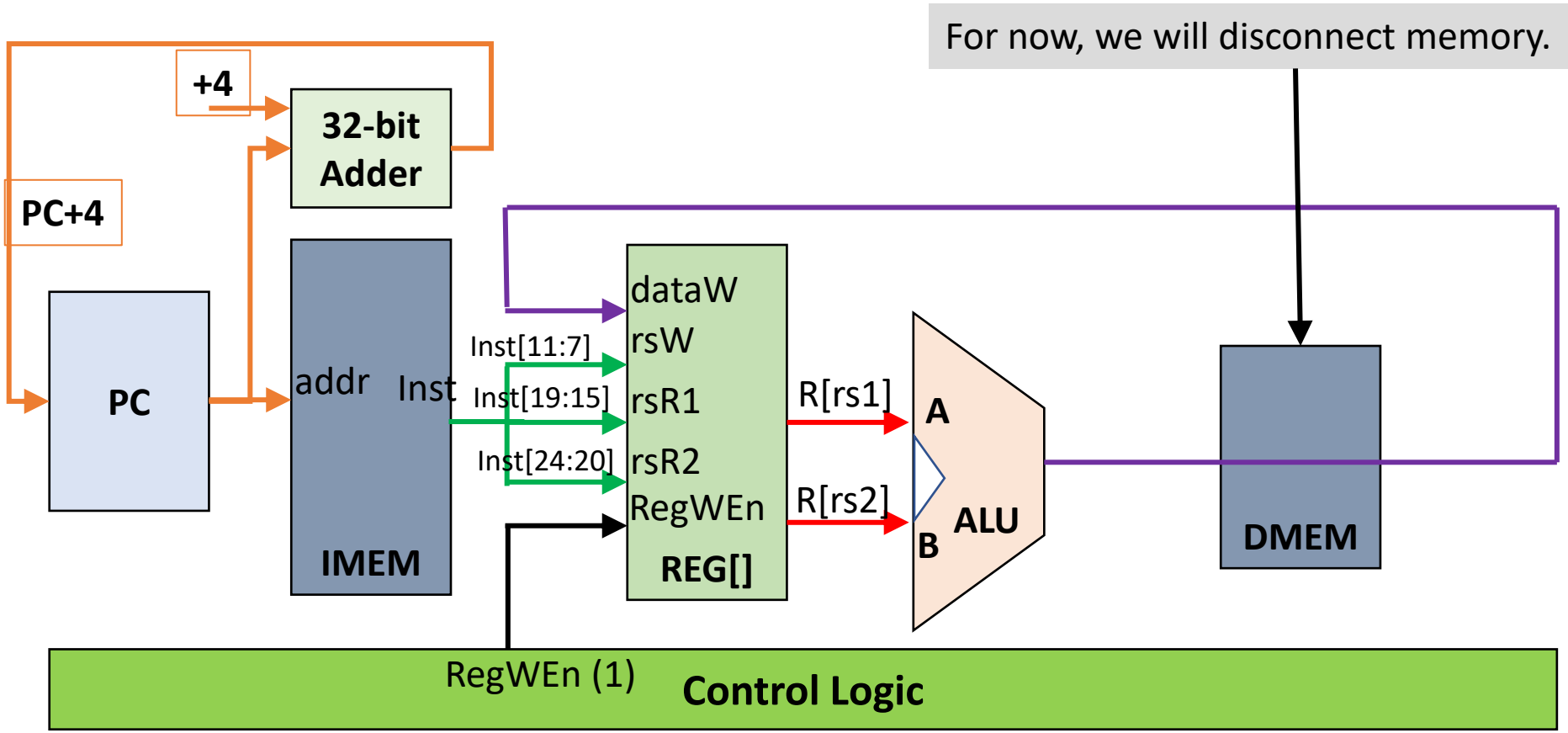
Write ALU output to destination register.



0000000	rs2	rs1	000	rd	0110011

- RegFile — Reg[rd] = Reg[rs1] + Reg[rs2]
- PC — PC = PC + 4

Create Full Datapath Step-by-Step



Implementing the sub Instruction

- Now we support two instructions in our RISC-V ISA
  - add/sub

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	
0000000	rs2	rs1	000	rd	0110011	**add**
0100000	rs2	rs1	000	rd	0110011	**sub**

sub rd, rs1, rs2

- Instruction bit inst[30] selects between add/sub
- Details left to control logic

sub is same as **add,** ALU subtracts operands instead of adding them

- RegFile Reg[rd] = Reg[rs1] - Reg[rs2]
- PC PC = PC + 4

Datapath for sub

0100000	rs2	rs1	000	rd	0110011

PC = PC + 4

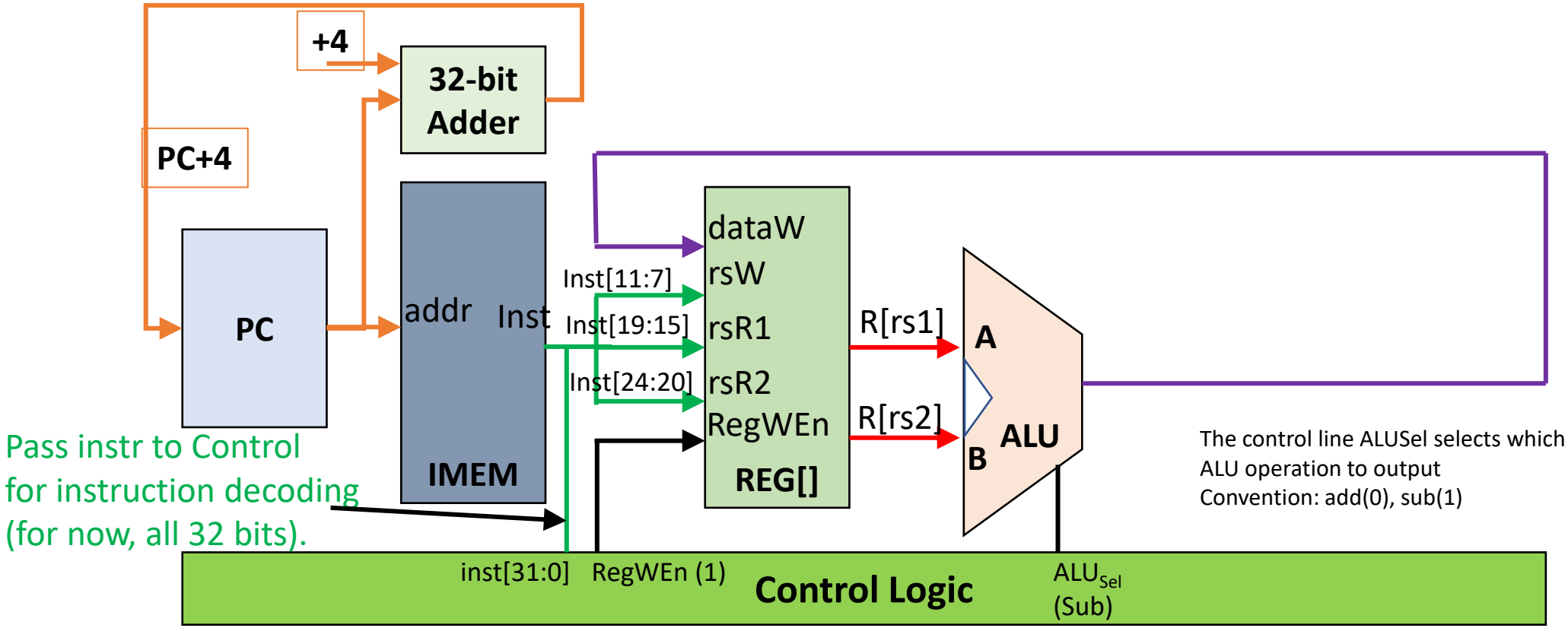
Increment PC to next instruction.

Split instruction to index into RegFile.

R[rd] = R[rs1] - R[rs2]

Feed read register values into ALU.

Write ALU output to destination register.



Universität Bremen

Supporting All R-Type Instructions

funct7	rs2	rs1	funct3	rd	opcode	
0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000000	rs2	rs1	001	rd	0110011	sll
0000000	rs2	rs1	010	rd	0110011	slt
0000000	rs2	rs1	011	rd	0110011	sltu
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	101	rd	0110011	srl
0100000	rs2	rs1	101	rd	0110011	sra
0000000	rs2	rs1	110	rd	0110011	or
0000000	rs2	rs1	111	rd	0110011	and

The Control Logic decodes funct3, funct7 instruction fields and selects appropriate ALU function by setting the control line ALU_{Sel}

Implementing the addi Instruction

- Let's add a new instruction
  - addi

$Imm_{11:0}$	rs1	funct3	rd	opcode
$Imm_{11:0}$	rs1	000	rd	0010011

I-Type

addi rd, rs1, imm

addi updates the same two states as before. But now we need to build **immediate imm**

- RegFile Reg[rd] = Reg[rs1] + **imm**
- PC PC = PC + 4

Datapath for addi

Exists

PC = PC + 4

Increment PC to next instruction.

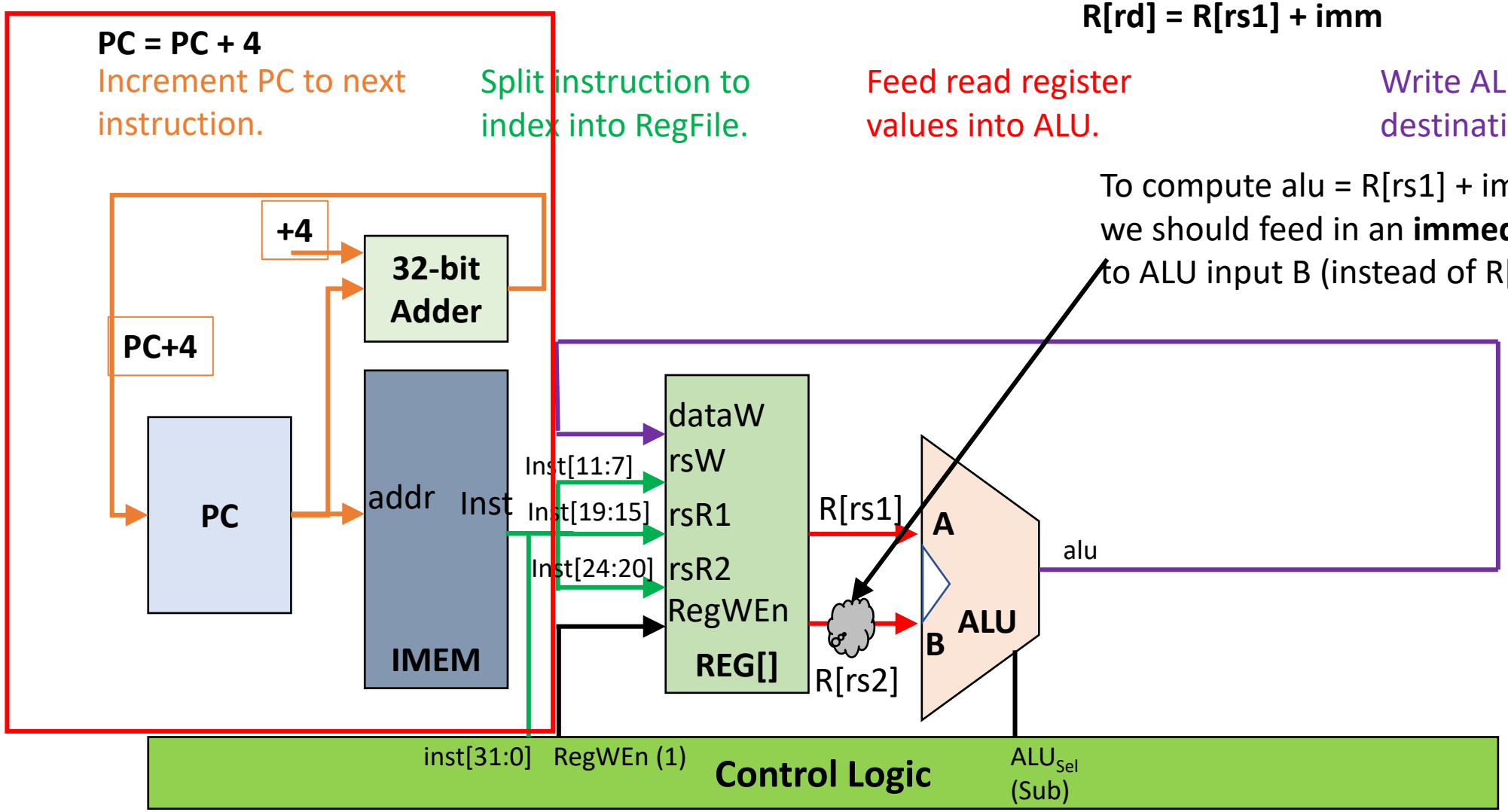
Split instruction to index into RegFile.

Feed read register values into ALU.

R[rd] = R[rs1] + imm

Write ALU output to destination register.

To compute alu = R[rs1] + imm we should feed in an **immediate** to ALU input B (instead of R[rs2]).



New Mux to Select Immediate for ALU

PC = PC + 4

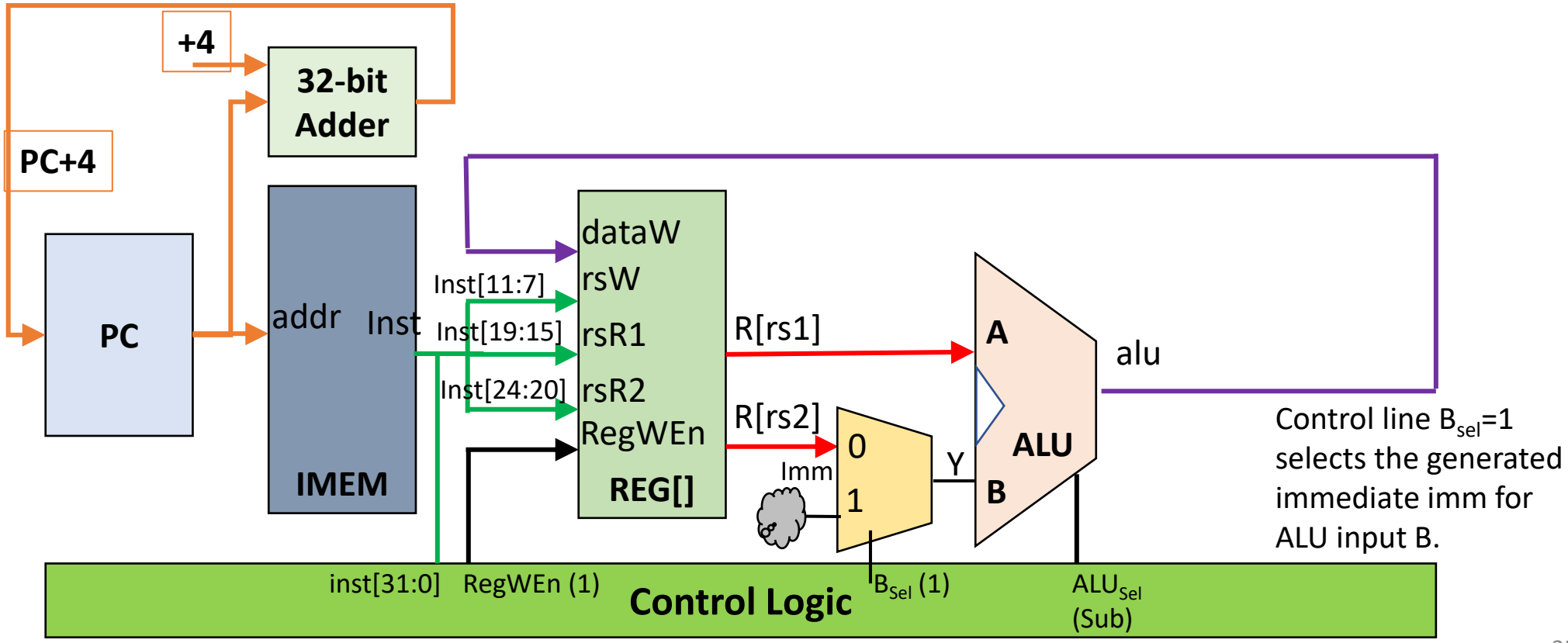
Increment PC to next instruction.

Split instruction to index into RegFile.

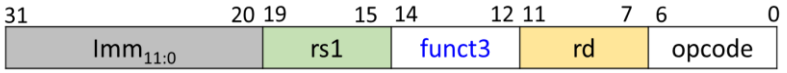
Feed read register values into ALU.

R[rd] = R[rs1] + imm

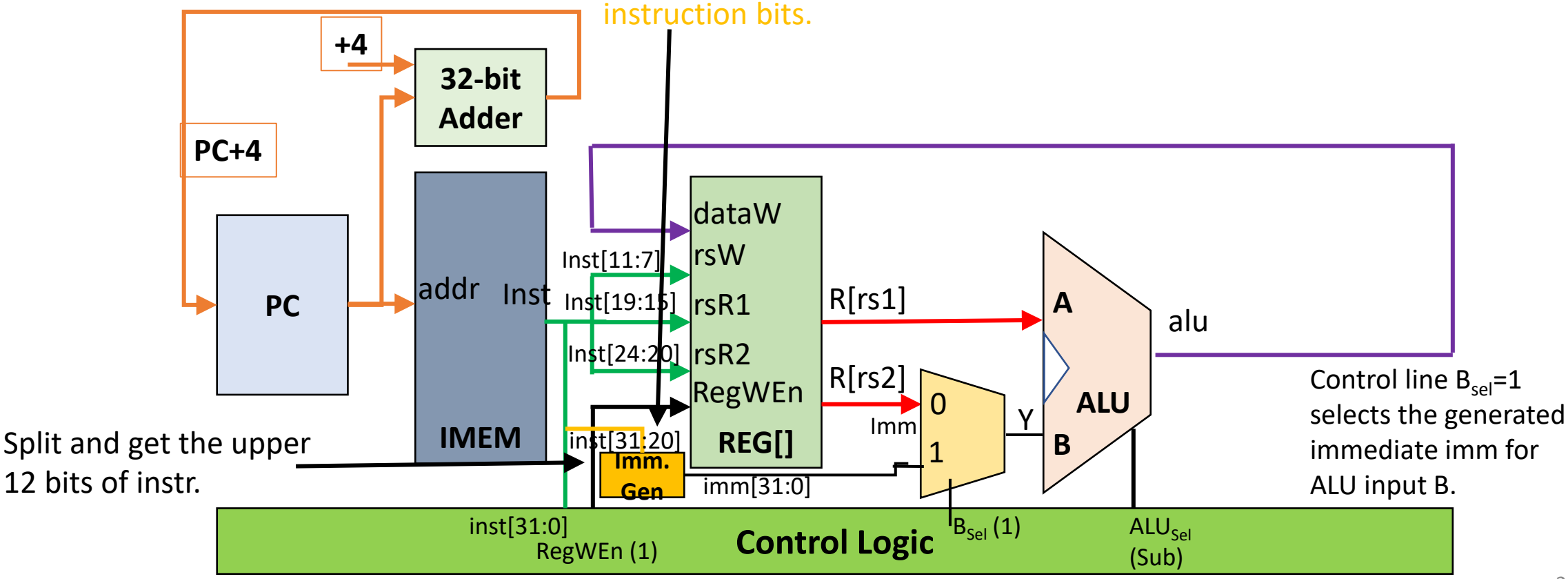
Write ALU output to destination register.



New Block to Generate 32-bit Immediate



Immediate Generation Block builds a 32-bit immediate imm from instruction bits.

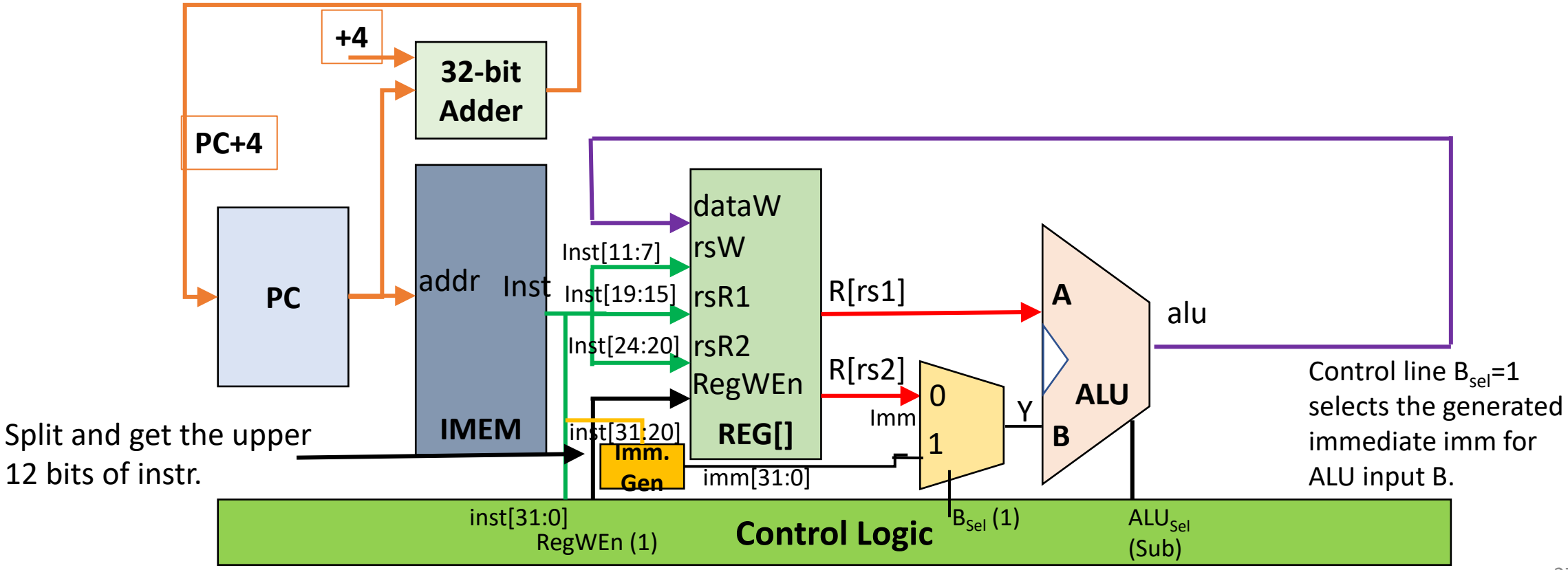


Split and get the upper 12 bits of instr.

Control line B_{sel}=1 selects the generated immediate imm for ALU input B.

Summary

- All data lines carry information
- Control logic determines what is useful/needed vs. what is ignored
  - e.g., ALUSel: chooses ALU operation; Bsel: chooses register/immediate for ALU input B.



Universität Bremen

RISC-V Processor Architecture

Prof. Dr. Rolf Drechsler
Dr. Muhammad Hassan
M.Sc. Jan Zielasko
M.Sc. Milan Funck