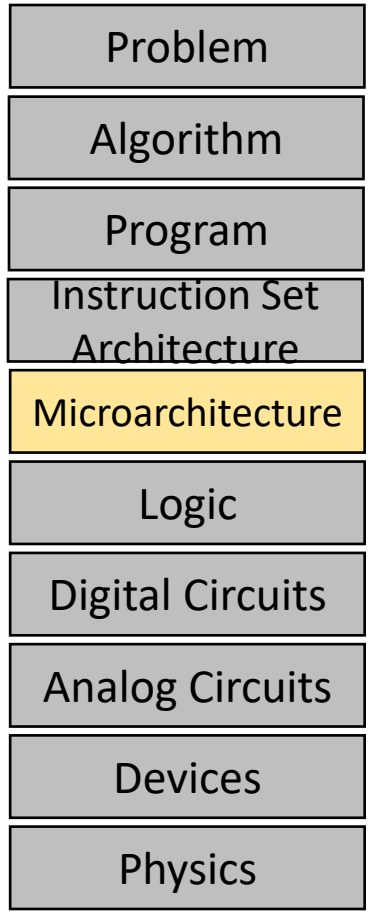
Universität Bremen

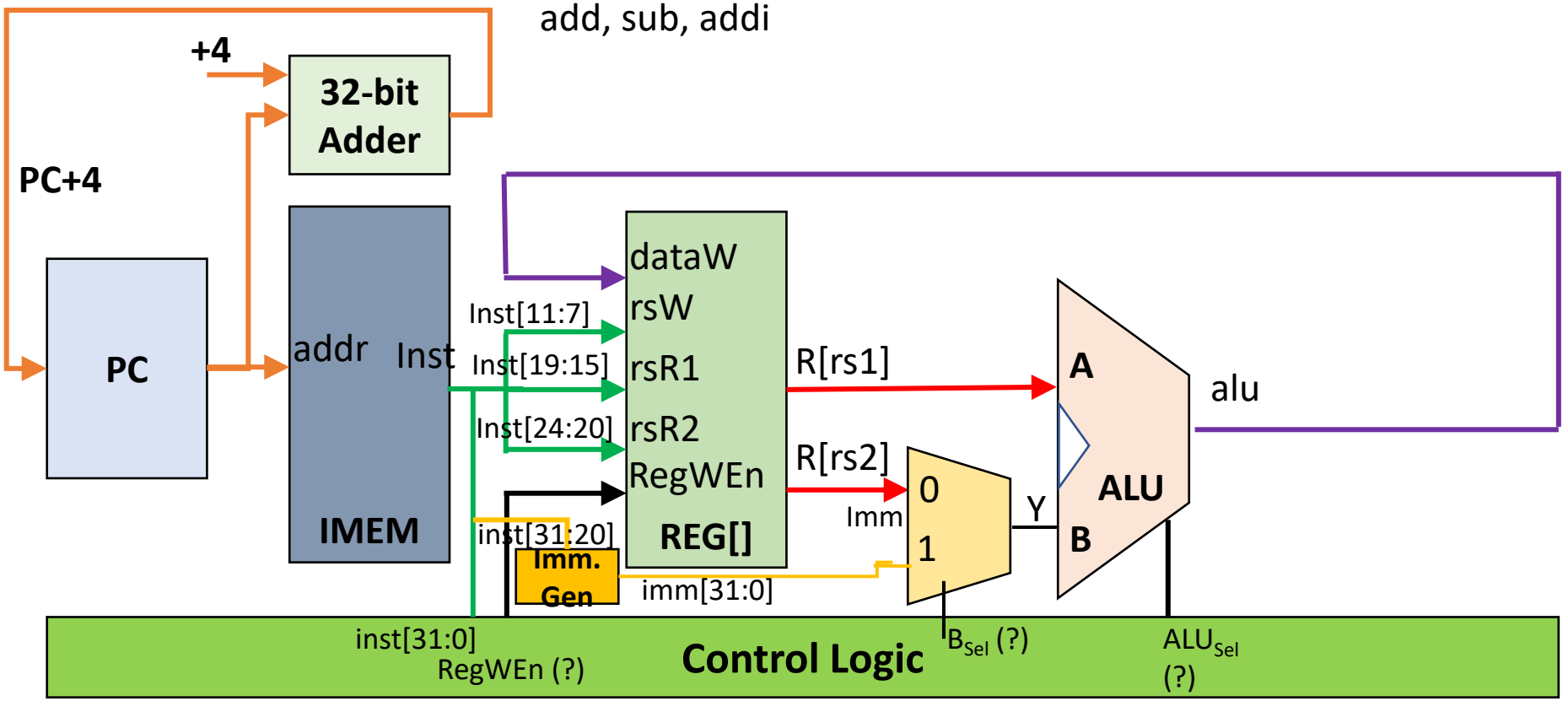
RISC-V Datapath cont...

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



Discussed the last time …

Instructions possible with this datapath:
add, sub, addi



Universität Bremen

Today's Agenda

- Implementing Load Word
  - Load Instructions
- Implementing Store Word
  - Store Instruction
- Implementing Branches
  - Branch Instructions
- Implementing Jumps
  - Jump Instruction
- Implementing Upper Immediate Instruction
- Implementing Controller
  - Logic-based implementation
  - ROM-based implementation

Universität Bremen

Today's Agenda

- Implementing Load Word
  - Load Instructions
- Implementing Store Word
  - Store Instruction
- Implementing Branches
  - Branch Instructions
- Implementing Jumps
  - Jump Instruction
- Implementing Upper Immediate Instruction
- Implementing Controller
  - Logic-based implementation
  - ROM-based implementation

Implementing the lw Instruction

- lw uses I-Format **lw x14, 8(x2)**

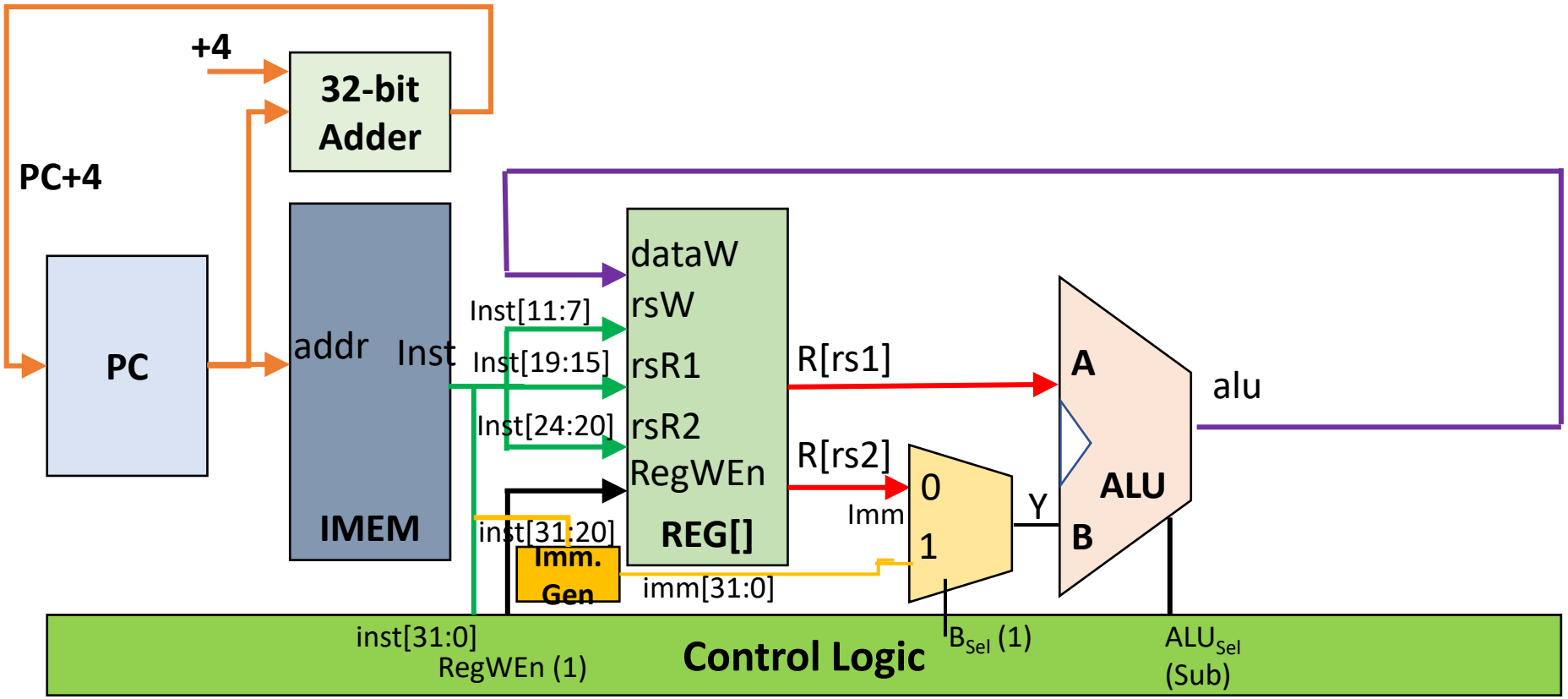
		Load word		Load	
Imm $_{11:0}$	rs1	funct3	rd	opcode	**I-Type**
000000001000	00010	010	01110	0000011	

- Similar datapath to **addi** but creates an address (not the final value stored)
- State element access now includes a memory read

addr = (Base register rs1)
+ (sign-extended **imm** offset)

- DMEM (read word at address addr)
- RegFile Reg[rs1] # read; Reg[rd] # write
- PC PC = PC + 4

Current Processor

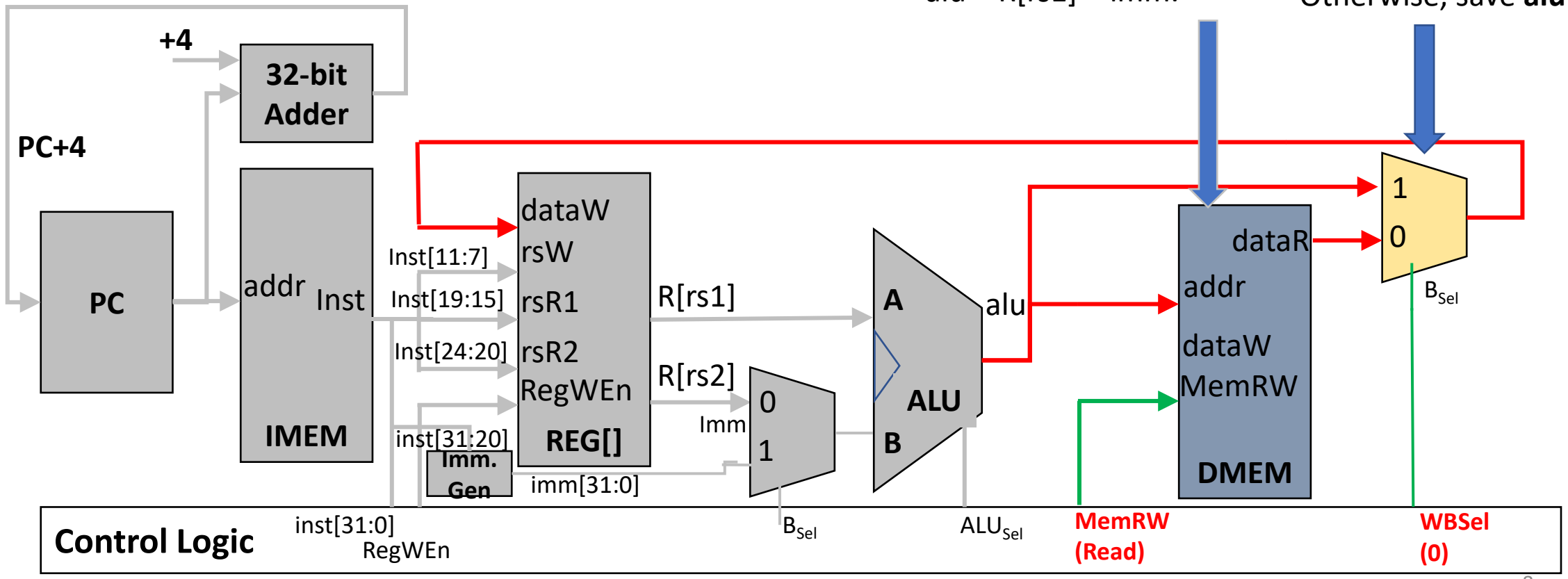


Saving Memory Read to RegFile

Imm $_{11:0}$	rs1	funct3	rd	opcode
000000001000	00010	010	01110	0000011

Read memory at address alu = R[rs1] + imm.

If load instruction, save **mem**. Otherwise, save **alu**.



Load Datapath

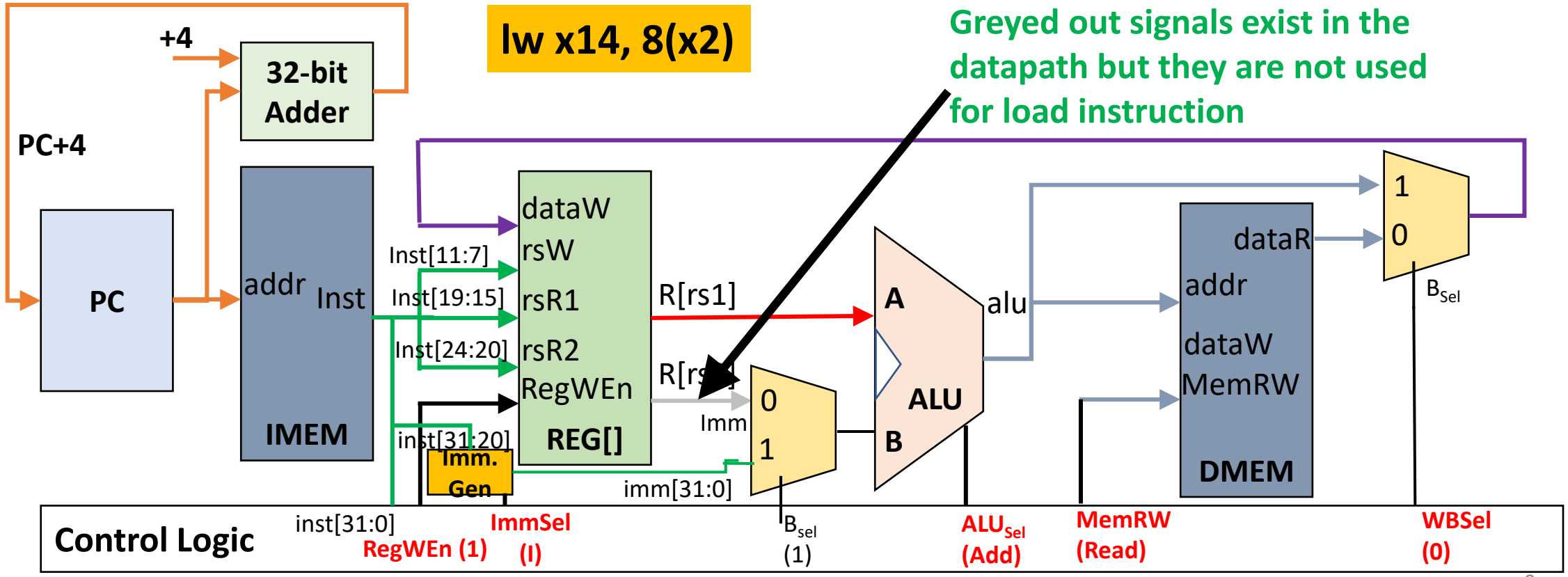
Increment PC to next instruction.

Immediate Generation Block builds a 32-bit immediate **imm**.

ALU computes address alu = R[rs1] + imm.

Read memory at address alu.

Write loaded memory value **mem** to register.



Q: How many stages of the datapath does Load use?

Increment PC to next instruction.

Immediate Generation Block builds a 32-bit immediate **imm**.

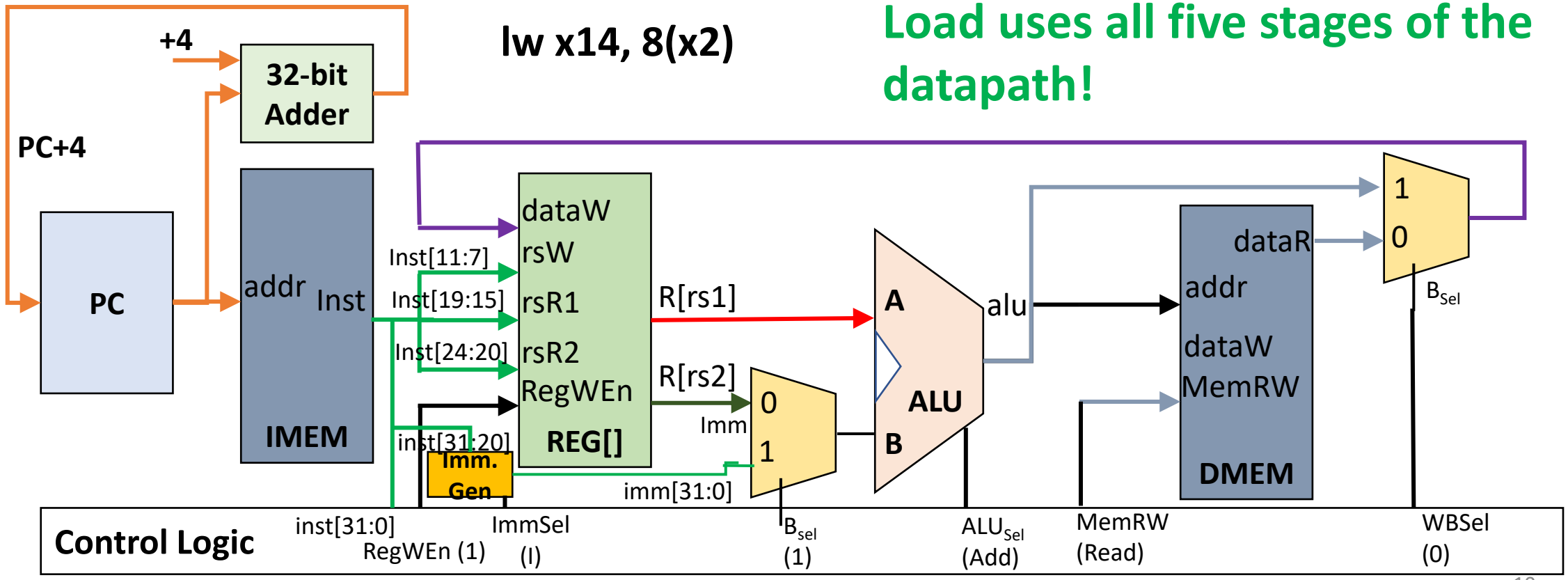
ALU computes address alu = R[rs1] + imm.

Read memory at address alu.

Write loaded memory value **mem** to register.

lw x14, 8(x2)

Load uses all five stages of the datapath!



Universität Bremen

RV32I Can Load Different Widths

	rs1	funct3	rd	opcode	
imm[11:0]	rs1	000	rd	0000011	lb
imm[11:0]	rs1	001	rd	0000011	lh
imm[11:0]	rs1	010	rd	0000011	lw
imm[11:0]	rs1	100	rd	0000011	lbu
imm[11:0]	rs1	101	rd	0000011	lhu

- To support narrower loads (lb, lh, lbu, lhu)
  - Load 32-bit word from memory;
  - Add additional logic to extract correct byte or halfword; and
  - Sign- or zero-extend result to 32 bits to write into RegFile.
  - Can be implemented with MUX + and a few gates.

- We don’t show this level of detail in lecture.
- Abstractions!

Universität Bremen

Today's Agenda

- Implementing Load Word
  - Load Instructions
- Implementing Store Word
  - Store Instruction
- Implementing Branches
  - Branch Instructions
- Implementing Jumps
  - Jump Instruction
- Implementing Upper Immediate Instruction
- Implementing Controller
  - Logic-based implementation
  - ROM-based implementation

Implementing the sw Instruction

- S-Format **sw x14, 36(x2)**

Store word STORE

$Imm_{11:5}$	rs2	rs1	funct3	$Imm_{4:0}$	opcode
0000001	01110	00010	010	00100	010011

- New Immediate Format 0000001 010
  - addr = (Base register rs1) + (sign-extended **imm** offset)
- State Elements Accessed
  - DMEM (write R[rs2] to word at address addr)
  - RegFile R[rs1] (base address), R[rs2] (value to store)
  - PC = PC + 4

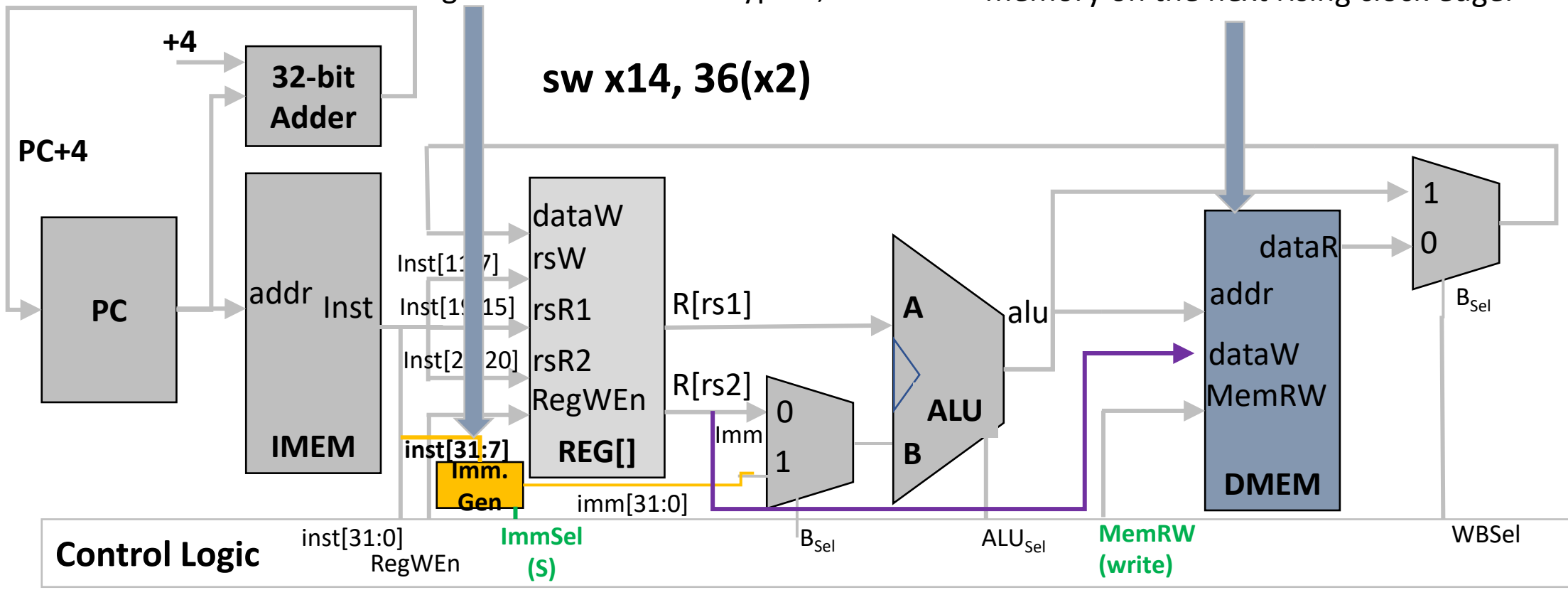
No RegFile write!

Adding sw Functionality

$Imm_{11:5}$	rs2	rs1	funct3	$Imm_{4:0}$	opcode
0000001	01110	00010	010	00100	010011

Control **ImmSel** selects how to generate immediate type: I, S.

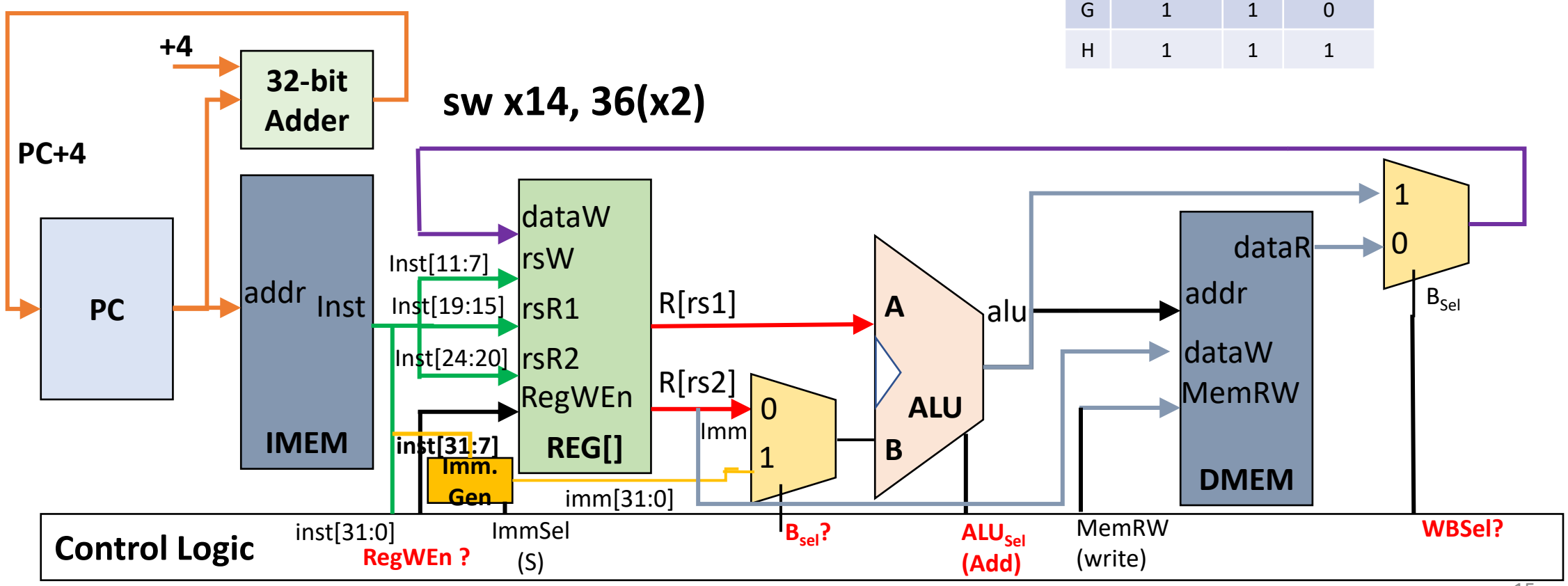
Control **MemRW=Write** saves rs2 to memory on the next rising clock edge.



Q:How to Set sw Control Lines?

Imm 11:5	rs2	rs1	010	Imm4:0	0100011

	RegWEn	B_{Sel}	WBSel
A	Read (0)	0	0
B	0	0	1
C	0	1	0
D	0	1	1
E	Write (1)	0	0
F	1	0	1
G	1	1	0
H	1	1	1



How to Set sw Control Lines?

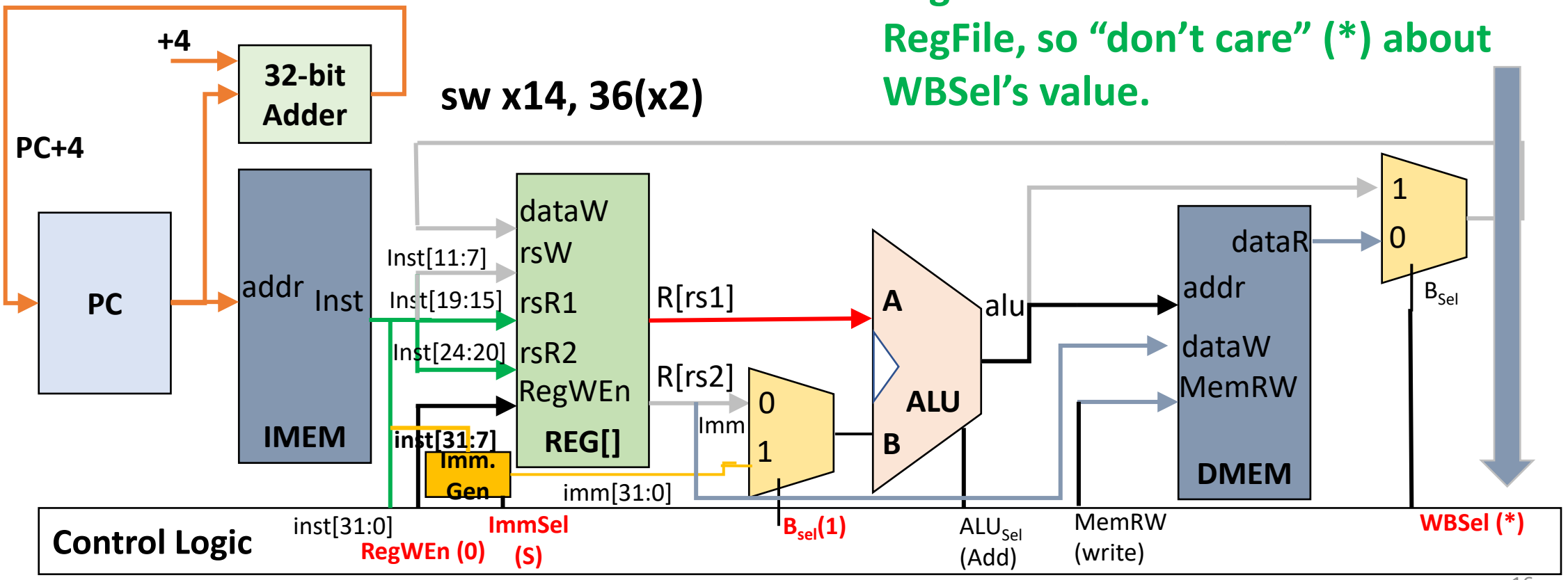
Increment PC to next instruction.

Build imm from S-type instruction.

ALU computes address alu = R[rs1] + imm.

Write memory at address alu.

RegWEn=0 means no write back to RegFile, so "don't care" (*) about WBSel's value.



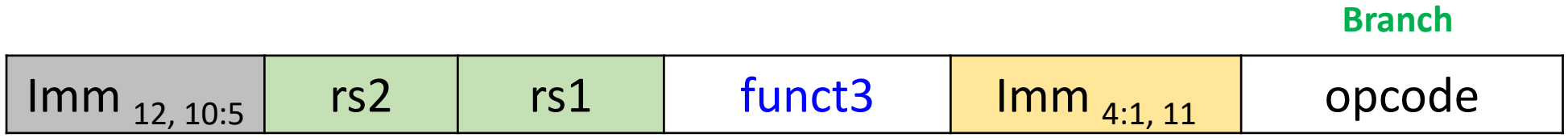
Universität Bremen

Today's Agenda

- Implementing Load Word
  - Load Instructions
- Implementing Store Word
  - Store Instruction
- Implementing Branches
  - Branch Instructions
- Implementing Jumps
  - Jump Instruction
- Implementing Upper Immediate Instruction
- Implementing Controller
  - Logic-based implementation
  - ROM-based implementation

B-Format – Conditional Branch

- B-Format (SB-Type) close to S-Format **opname rs1, rs2, Label**



- New Immediate Format

- PC state element **conditionally** changes
  - RegFile R[rs1], R[rs2] (read only, for branch comparison)
  - PC PC = PC + imm (if branch taken)
    PC = PC + 4 (otherwise, not taken)

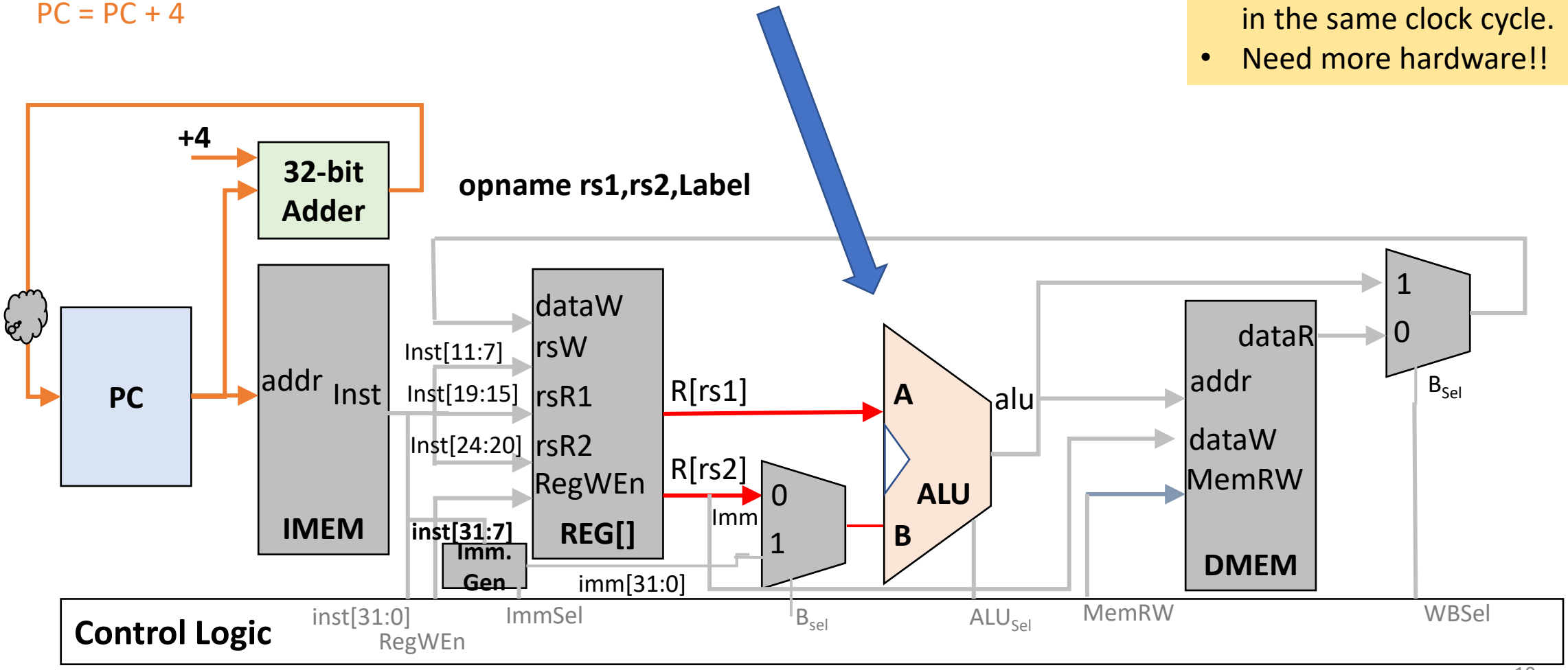
What do Branches Need to Compute?

PC = PC + imm ??

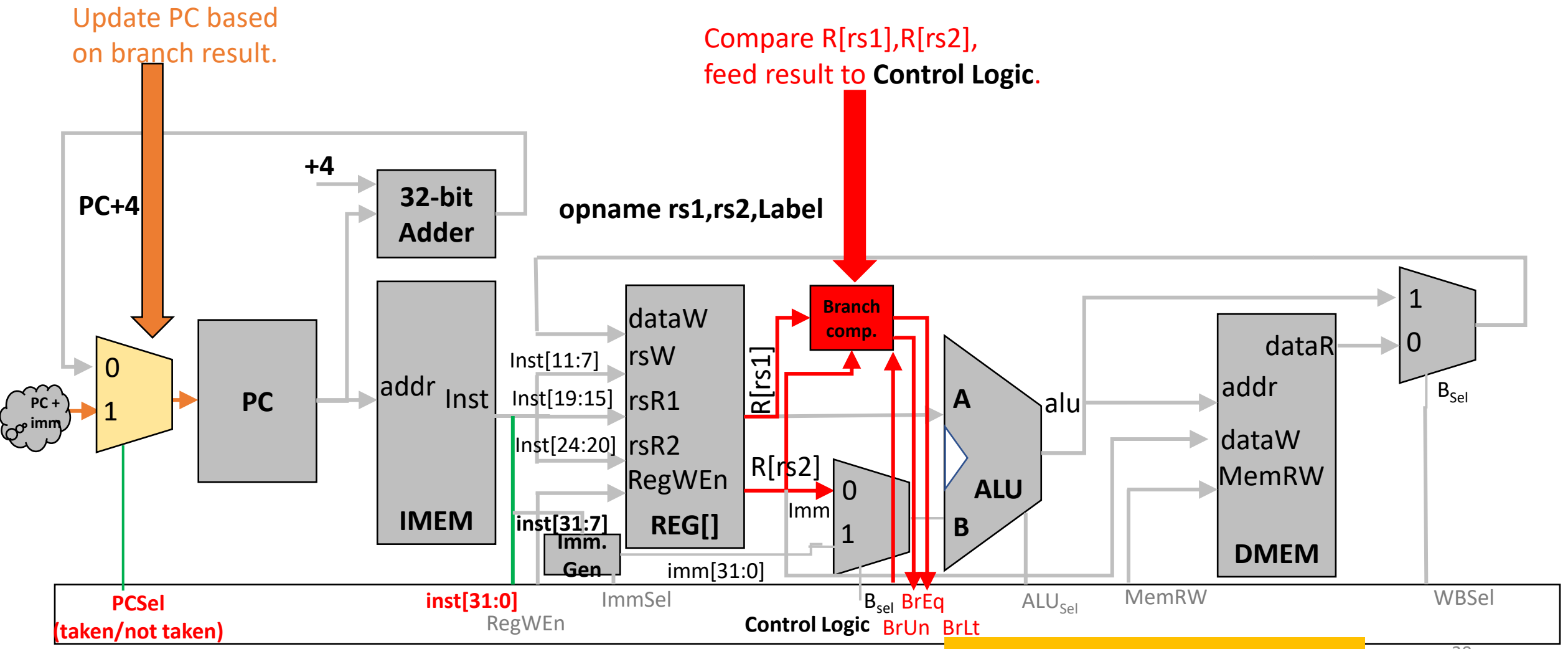
PC = PC + 4

Compare R[rs1] , R[rs2] ??

- Only one ALU accessible in the same clock cycle.
- Need more hardware!!



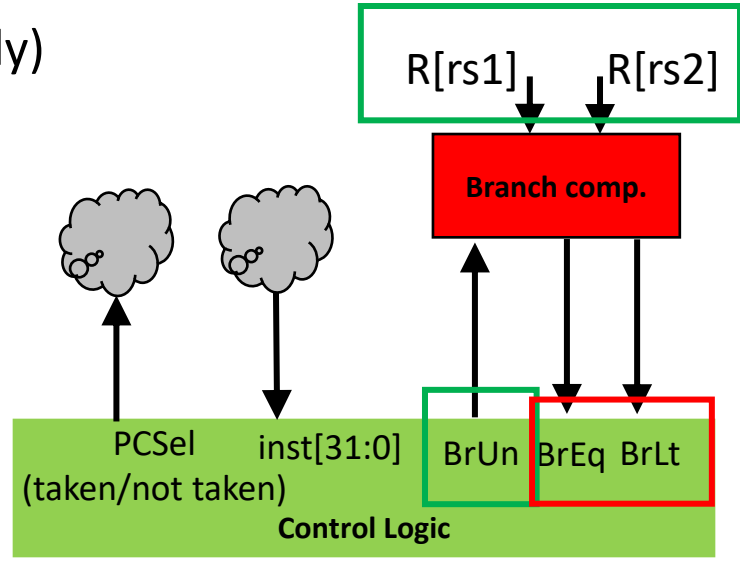
The Branch Comparator Block



BrUn: Control; BrLT, BrEq: Data

The Branch Comparator Block

- The branch comparator is a combinational logic block
  - Input:
    - Two data busses A and B (datapath R[rs1] and R[rs2], respectively)
    - **BrUn** (“Branch Unsigned”) control bit
  - Output:
    - **BrEq** flag: 1 if A == B
    - **BrLT** flag: 1 if A < B.
    - Unsigned comparison if **BrUn=1**, signed otherwise.
- Control Logic
  - Set **BrUn** based on current instruction, inst[31:0].
  - Set **PCSel** based on branch flags **BrLT**, **BrEq**.



Examples

- blt:
  - If **BrLT**=1 and **BrEq**=0, then **PCSel**=taken.
- bge: $(A \geq B) = \overline{A < B}$
  - If **BrLT**=0, then **PCSel**=taken.

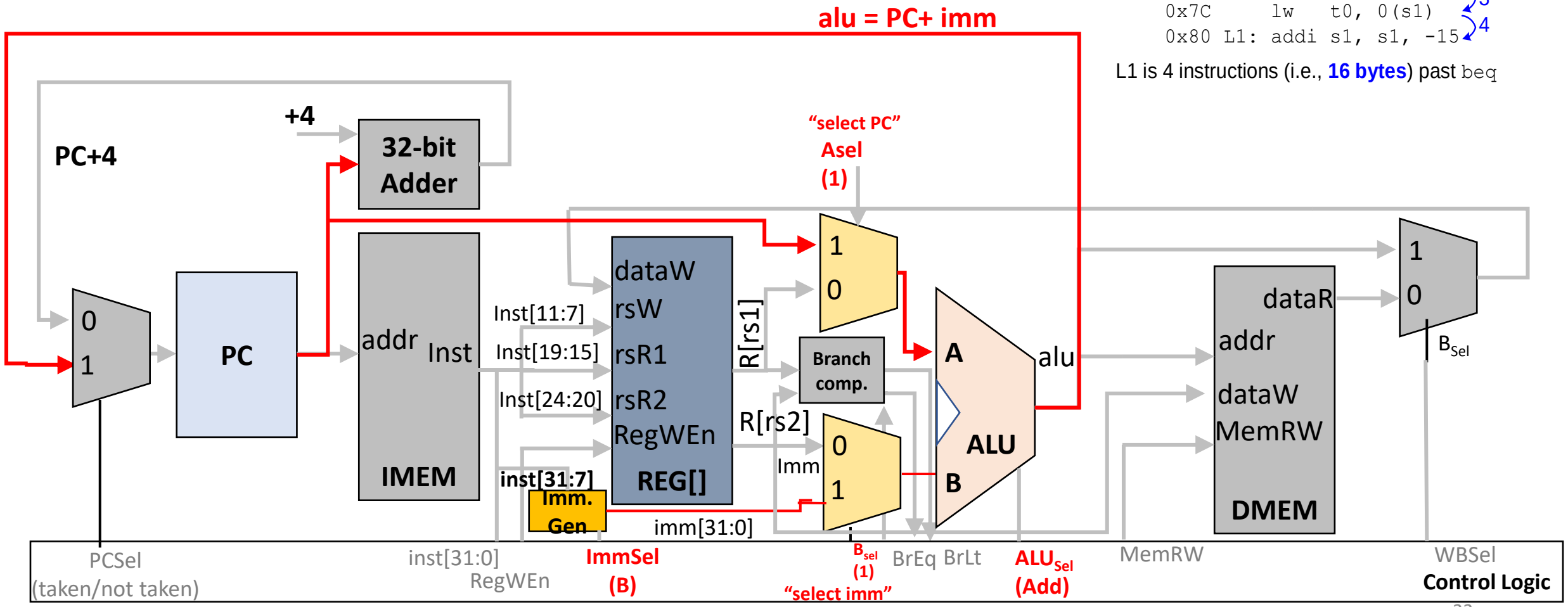
The ALU Computes PC+ Imm

opname rs1,rs2,Label

To compute the address to branch to, use the ALU

```
# RISC-V Assembly
0x70       beq  s0, t5, L1
0x74       add  s1, s2, s3
0x78       sub  s5, s6, s7
0x7C       lw   t0, 0(s1)
0x80 L1:   addi s1, s1, -15
```

L1 is 4 instructions (i.e., **16 bytes**) past `beq`



Branch Datapath

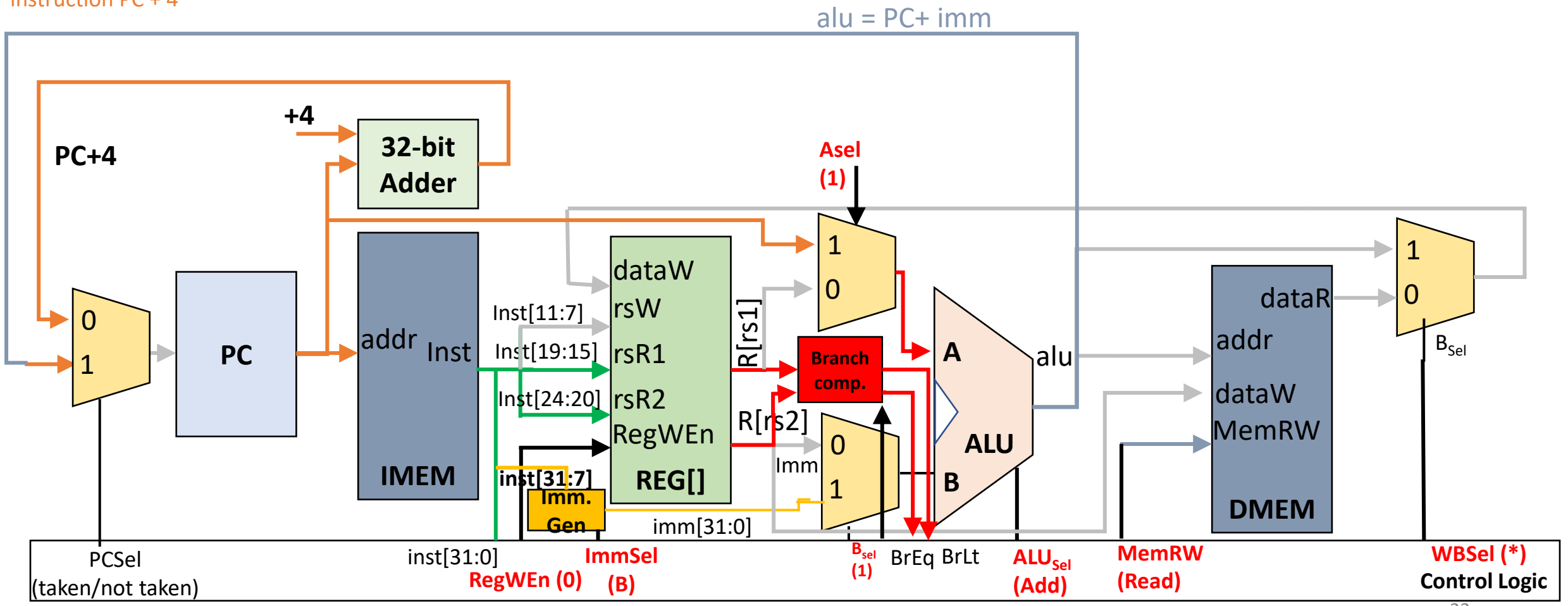
If **PCSel**=taken, update PC to ALU output. Else, update to next instruction PC + 4

Build imm from B-type instruction

- Compute branch; feed to **Control**
- Compute PC + imm

Don't write to memory

Don't write to registers



Universität Bremen

Today's Agenda

- Implementing Load Word
  - Load Instructions
- Implementing Store Word
  - Store Instruction
- Implementing Branches
  - Branch Instructions
- Implementing Jumps
  - Jump Instruction
- Implementing Upper Immediate Instruction
- Implementing Controller
  - Logic-based implementation
  - ROM-based implementation

J-Format – Unconditional Jump

- J-Format **jal rd, Label**

J-Type

- New Immediate Format
- State elements updated
  - PC PC = PC + imm (unconditional PC-relative jump)
  - RegFile rd = PC + 4

Save return address to RegFile destination register.

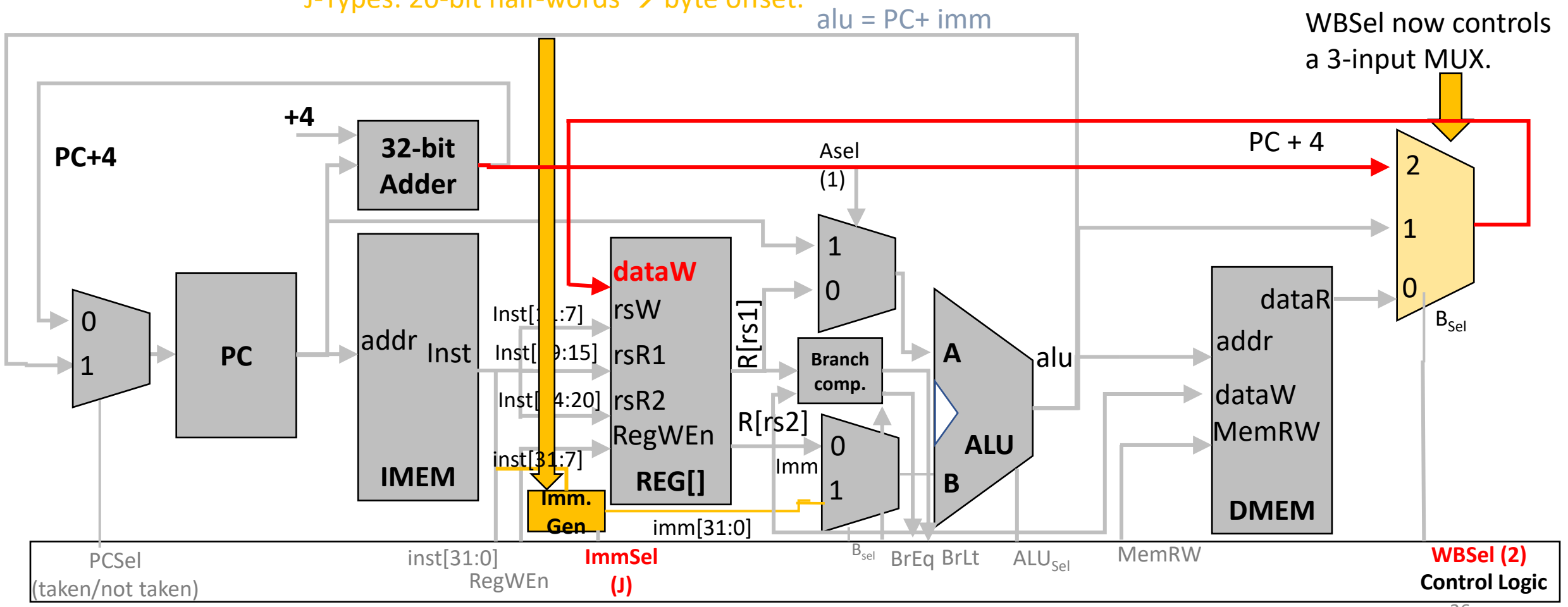
Block Updates for JAL

PC = PC + imm
rd = PC + 4

Save return address to RegFile destination register.

Immediate Generation Block needs to support J-Types: 20-bit half-words → byte offset.

To save rd = PC + 4, WBSel now controls a 3-input MUX.



PC = PC + imm
rd = PC + 4

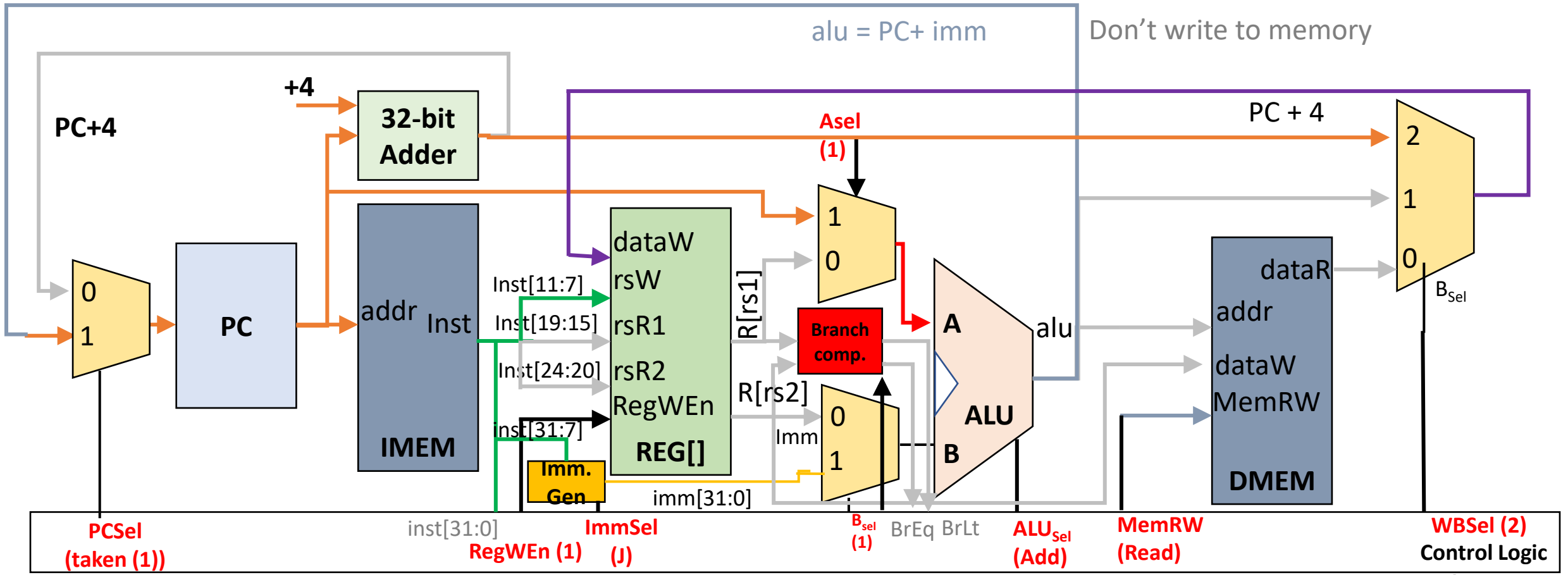
JAL Datapath

- Feed PC into blocks.
- Write ALU output to PC.

Generate byte offset imm for 20-bit PC-relative jump.

Compute PC + imm.

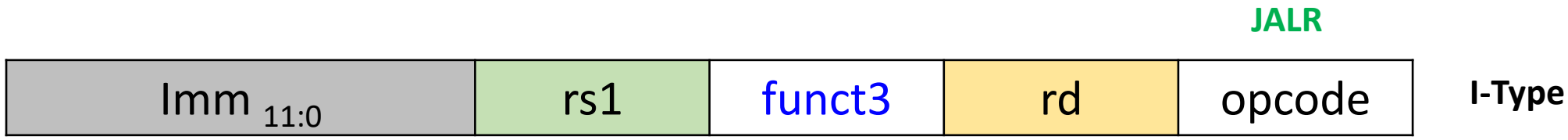
Write PC + 4 to destination register.



For JAL, "don't care" about branch.

I-Format Instruction Layout - jalr

- jalr uses I-Format **jalr rd,rs1,imm**



  - Two changes to state
    - PC PC = R[rs1] + imm (absolute addressing)
    - RegFile R[rd] = PC + 4

- I-Format means jalr uses the same immediates as arithmetic/loads
  - In other words, imm is already a byte offset.

Control ImmSel is based on instruction format, not instruction. So far: I,S,B,J

PC = PC + imm
rd = PC + 4

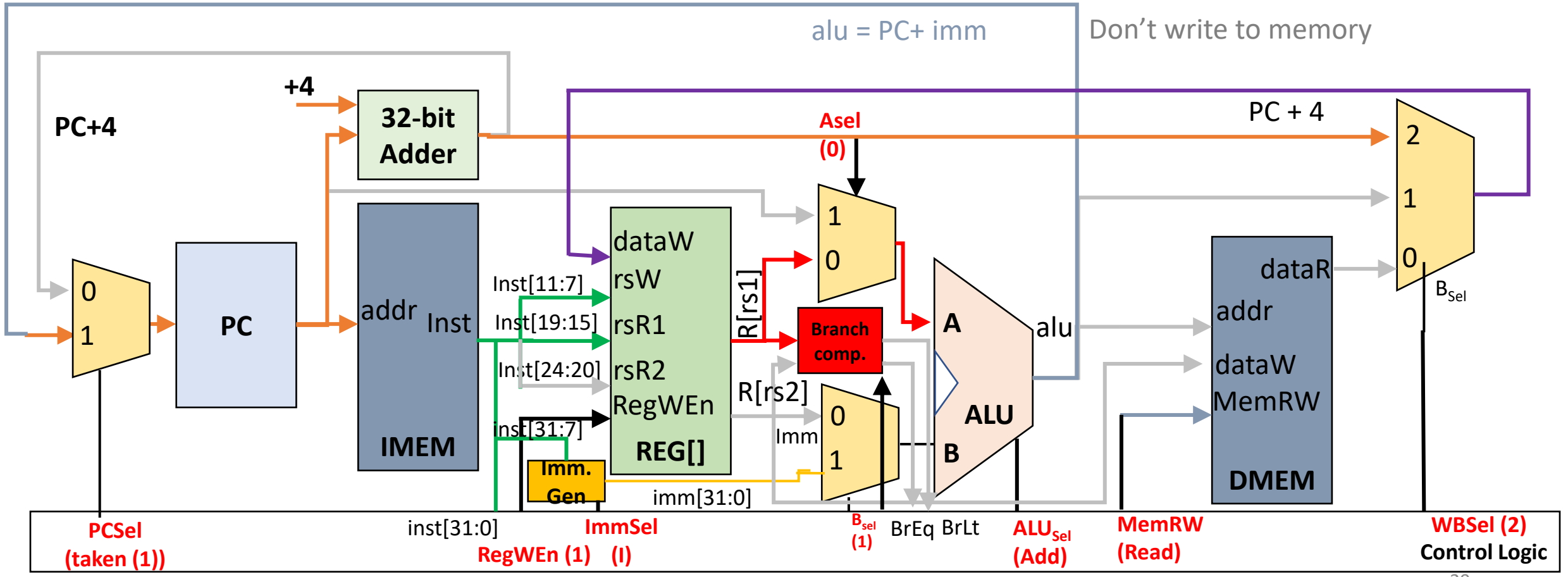
JALR Datapath

- Feed PC into blocks.
- Write ALU output to PC.

Generate 12-bit imm (I-Format)

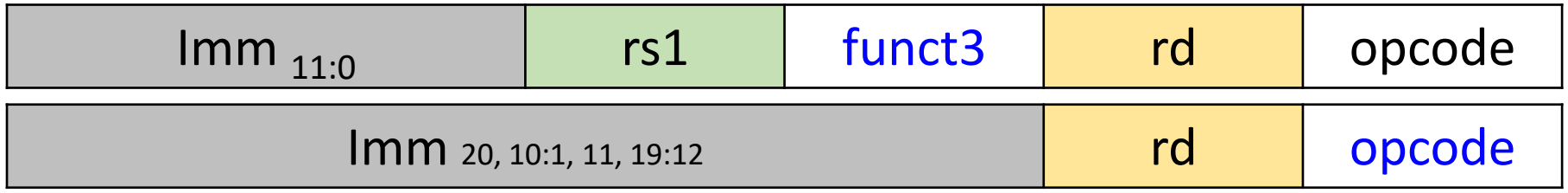
Compute rs1 + imm.

Write PC + 4 to destination register.



Universität Bremen

JAL vs JALR



Details	JAL	JALR
Jump Target	PC-relative (immediate offset)	Indirect (via a register + offset)
Offset Size	21-bit signed immediate	12-bit signed immediate.
Use Case	Direct jumps (e.g., subroutine calls).	Indirect jumps (e.g., returns, dynamic targets).

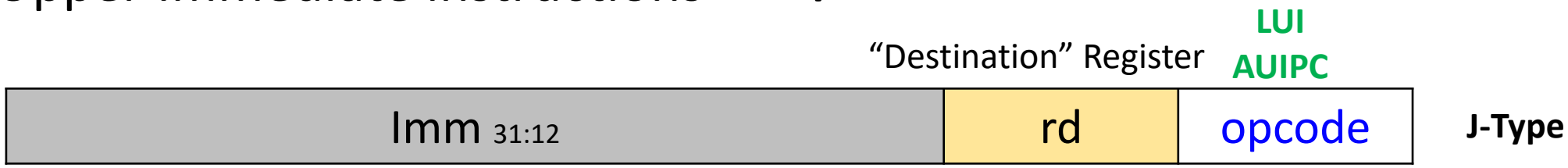
Universität Bremen

Today's Agenda

- Implementing Load Word
  - Load Instructions
- Implementing Store Word
  - Store Instruction
- Implementing Branches
  - Branch Instructions
- Implementing Jumps
  - Jump Instruction
- Implementing Upper Immediate Instruction
- Implementing Controller
  - Logic-based implementation
  - ROM-based implementation

U-Format Instruction Layout

- Upper Immediate instructions **opname rd,immed**



Immediate represents upper 20 bits of a 32-bit immediate imm.

- New Immediate Format
- Used for two instructions
  - lui : Load Upper Immediate
  - auipc : Add Upper Immediate to PC
  - Both increment PC to next instruction and save to destination register.

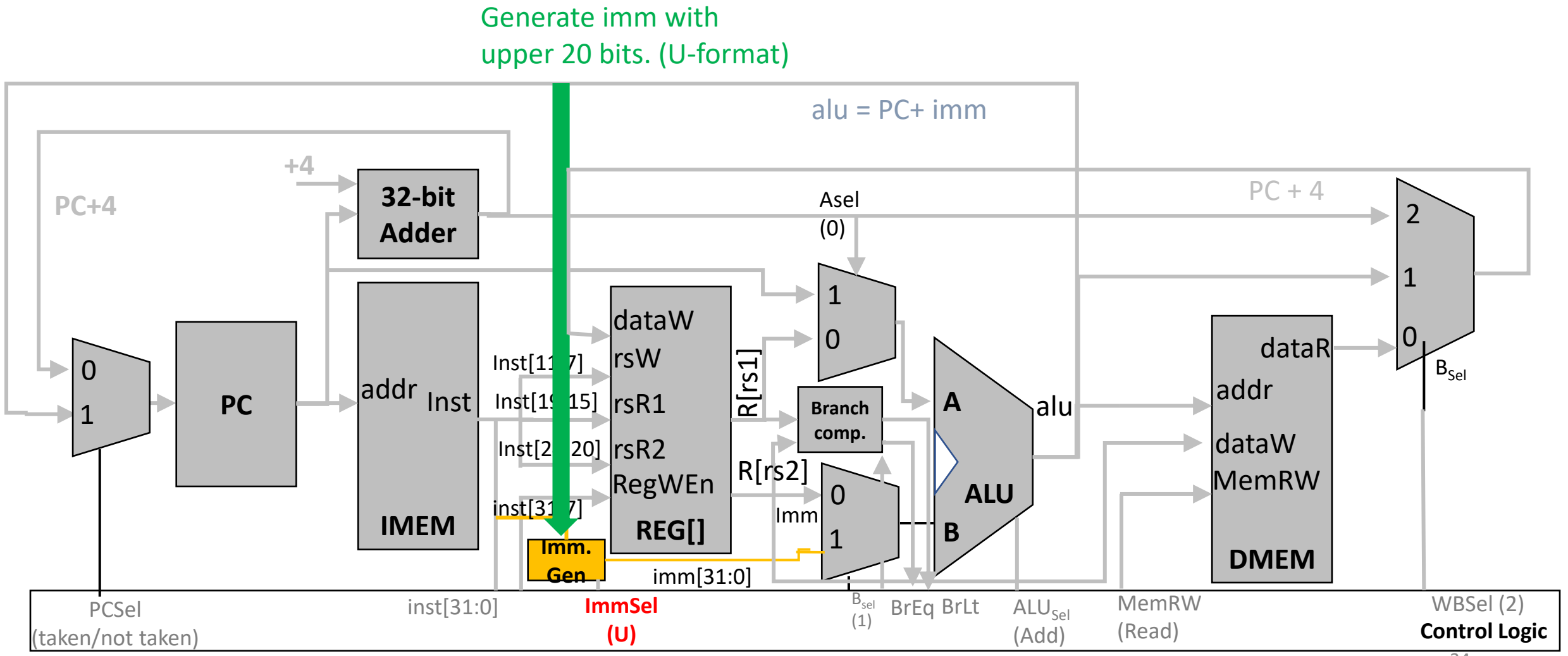
LUI Instruction

- Load Upper Immediate
  - **LUI rd, imm**
  - load a 20-bit immediate value into the upper 20 bits of a 32-bit register
  - set the lower 12 bits to zero
- construction of larger immediate values or absolute addresses
  - Combined with ADDI or other instructions, you can construct any 32-bit constant

```
LUI x5, 0x12345     # Load 0x12345 into x5
ADDI x5, x5, 0x678  # Add 0x678 to x5 (now x5 = 0x12345678)
```

- setting up base addresses for memory access (e.g., global variables, large data structures)

U-Format Block Update: Immediates



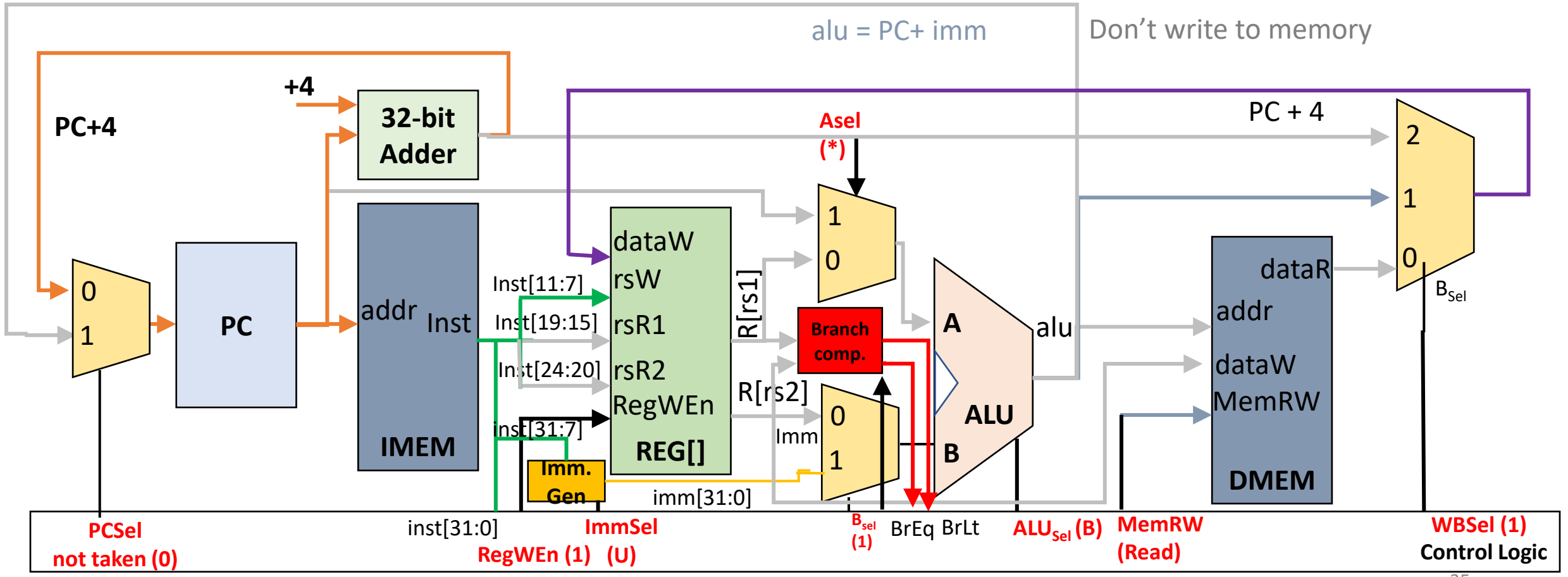
LUI Datapath

- Increment PC to next instruction

Generate imm with upper 20-bits (U-Format)

Grab only imm (ALU_{sel} = B)

Write result to destination register.



AUIPC Instruction

- Add Upper Immediate to PC
- **AUIPC rd, imm**
  - used to compute a Program Counter (PC)-relative address
  - adds a 20-bit upper immediate to the current PC value
  - stores the result in a destination register
- Facilitates position-independent code execution by referencing memory or data relative to the current location in the program

```
AUIPC x5, 0x12345   # x5 = PC + 0x12345
ADDI x5, x5, 0x678  # x5 = x5 + 0x678
```

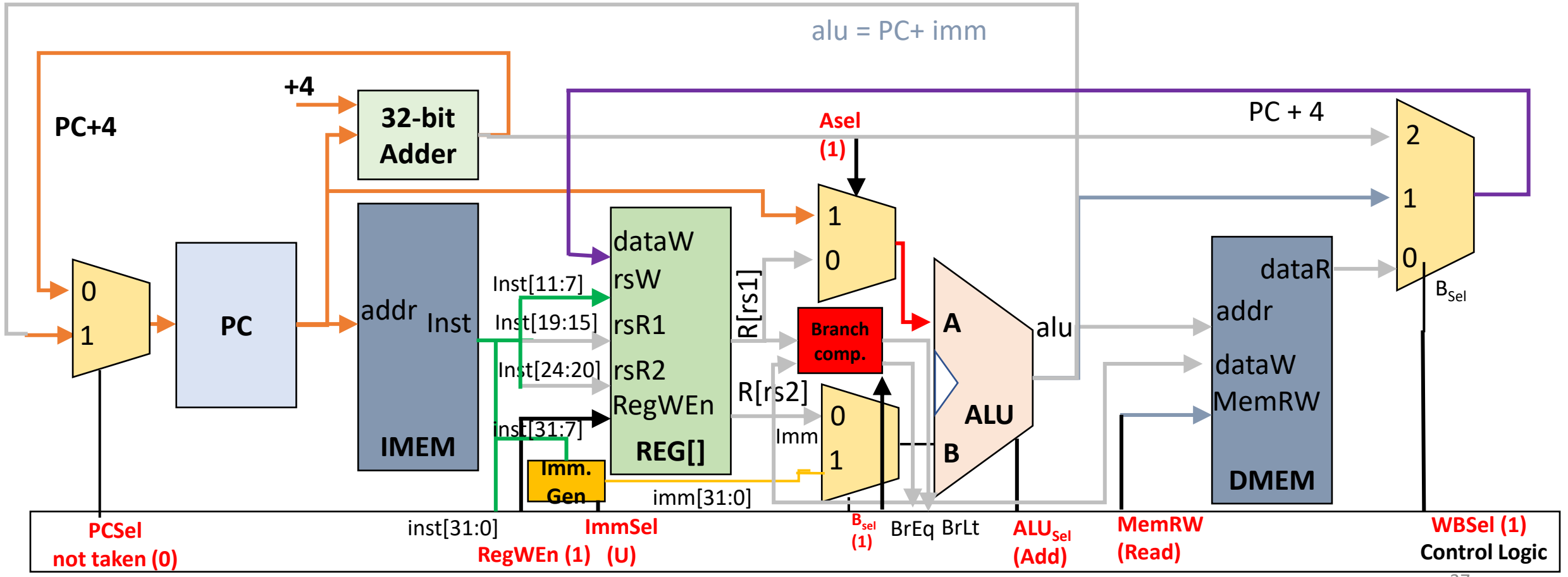
AUIPC Datapath

- Increment PC to next instruction

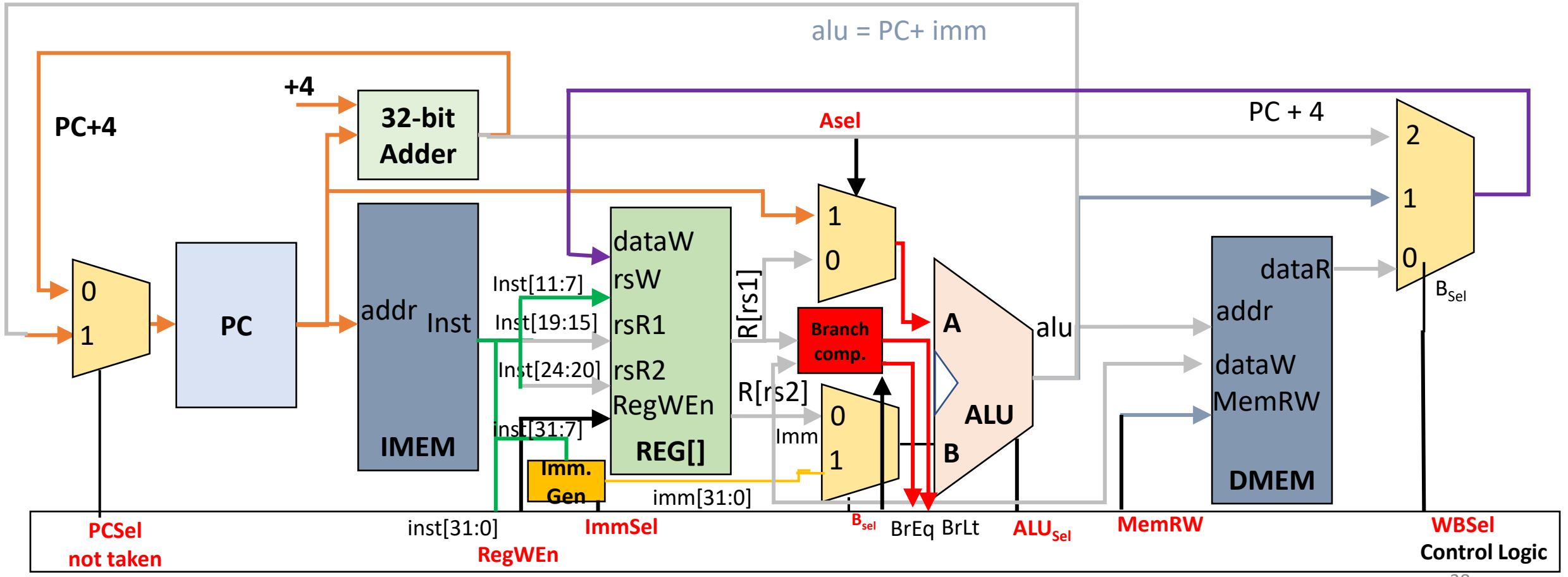
Generate imm with upper 20-bits (U-Format)

Add PC + imm

Write result to destination register.



Datapath


alu = PC+ imm
PC+4
+4
32-bit Adder
Asel
PC + 4
0
1
PC
addr
Inst
IMEM
Inst[11:7]
Inst[19:15]
Inst[24:20]
inst[31:7]
dataW
rsW
rsR1
rsR2
RegWEn
REG[]
R[rs1]
R[rs2]
Imm.
Gen
imm[31:0]
Imm
1
0
Branch comp.
A
ALU
B
alu
2
1
0
B_{Sel}
dataR
addr
dataW
MemRW
DMEM
PCSel
not taken
inst[31:0]
RegWEn
ImmSel
B_{sel}
BrEq
BrLt
ALU_{Sel}
MemRW
WBSel
Control Logic

Universität Bremen

Today's Agenda

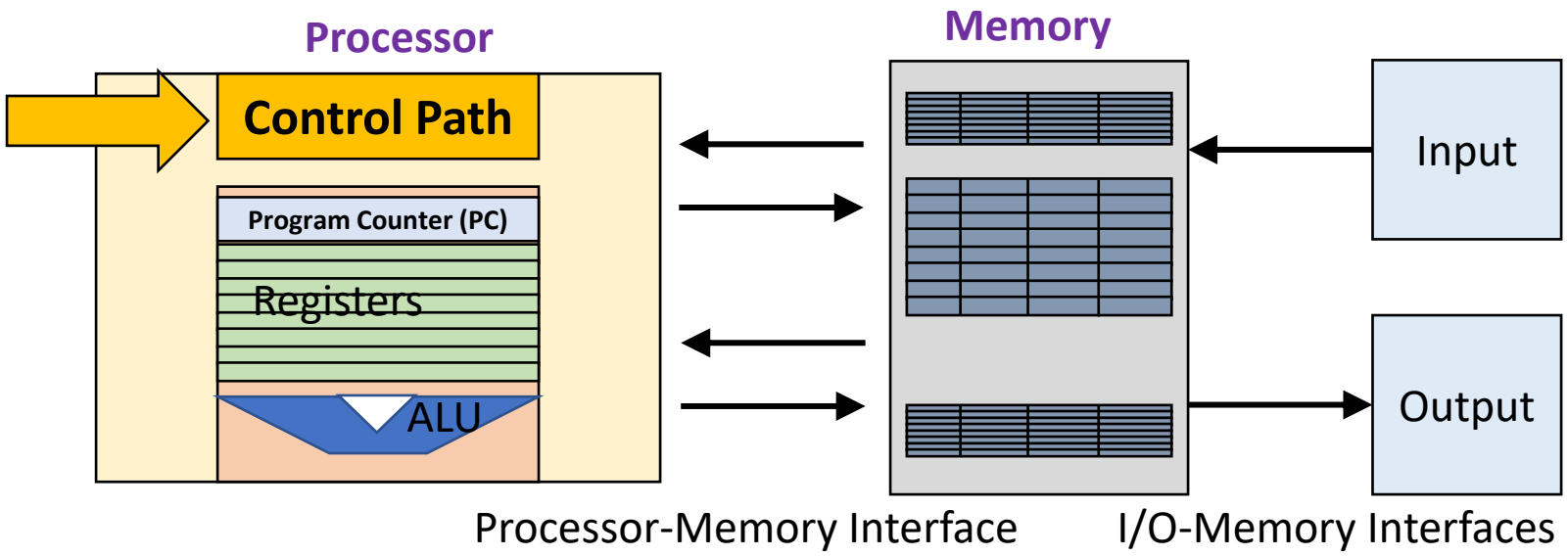
- Implementing Load Word
  - Load Instructions
- Implementing Store Word
  - Store Instruction
- Implementing Branches
  - Branch Instructions
- Implementing Jumps
  - Jump Instruction
- Implementing Upper Immediate Instruction
- Implementing Controller
  - Logic-based implementation
  - ROM-based implementation

Universität Bremen

Controller

- We have designed a complete datapath
  - Capable of executing all RISC-V instructions in one cycle each
  - Not all units (hardware) used by all instructions
- 5 Phases of execution
  - IF, ID, EX, MEM, WB
  - Not all instructions are active in all phases
- Controller specifies how to execute instructions
  - We still need to design it

Assembly Variables – Registers



Universität Bremen

Control Logic Truth Table

Inst[31:0]	BrEq	BrLT	PCSel	ImmSel	BrUN	ASel	BSel	ALUSel	MemRW	RegWEn	WBSel
add	*	*	+4	*	*	Reg	Reg	Add	Read	1	ALU
sub	*	*	+4	*	*	Reg	Reg	Sub	Read	1	ALU
r-r op	*	*	+4	*	*	Reg	Reg	(Op)	Read	1	ALU
addi	*	*	+4	I	*	Reg	Imm	Add	Read	1	ALU
lw	*	*	+4	I	*	Reg	Imm	Add	Read	1	Mem
sw	*	*	+4	S	*	Reg	Imm	Add	Write	0	*
beq	0	*	+4	B	*	PC	Imm	Add	Read	0	*
beq	1	*	ALU	B	*	PC	Imm	Add	Read	0	*
bne	0	*	ALU	B	*	PC	Imm	Add	Read	0	*
jalr	*	*	ALU	I	*	Reg	Imm	Add	Read	1	PC+4
jal	*	*	ALU	J	*	PC	Imm	Add	Read	1	PC+4

Universität Bremen

Control Realization Options

- ROM
  - “Read-Only Memory”
  - Regular structure
  - Can be easily reprogrammed
    - fix errors
    - add instructions
  - Popular when designing control logic manually
- Combinatorial Logic
  - Today, chip designers use logic synthesis tools to convert truth tables to networks of gates

RV32I, A Nine-bit ISA!

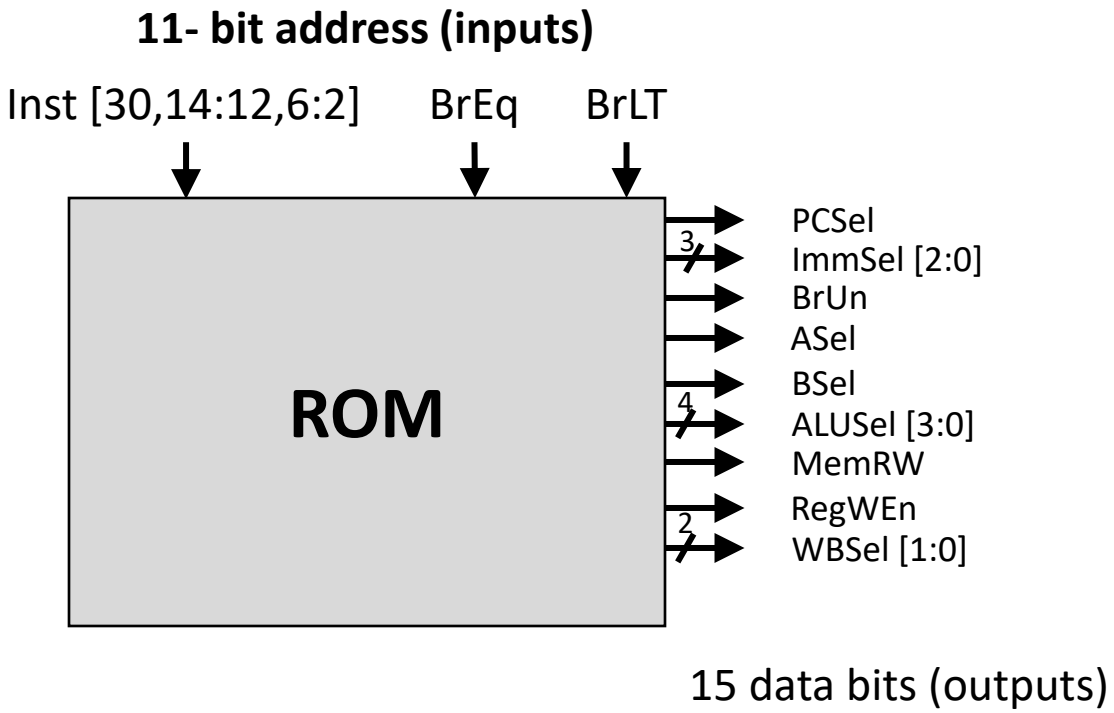
- Instruction type encoded using only 9 bits:
  - inst[6:2]
  - inst[14:12]
  - inst[30]

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
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
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

https://five-embeddev.com/riscv-isa-manual/latest/instr-table.html

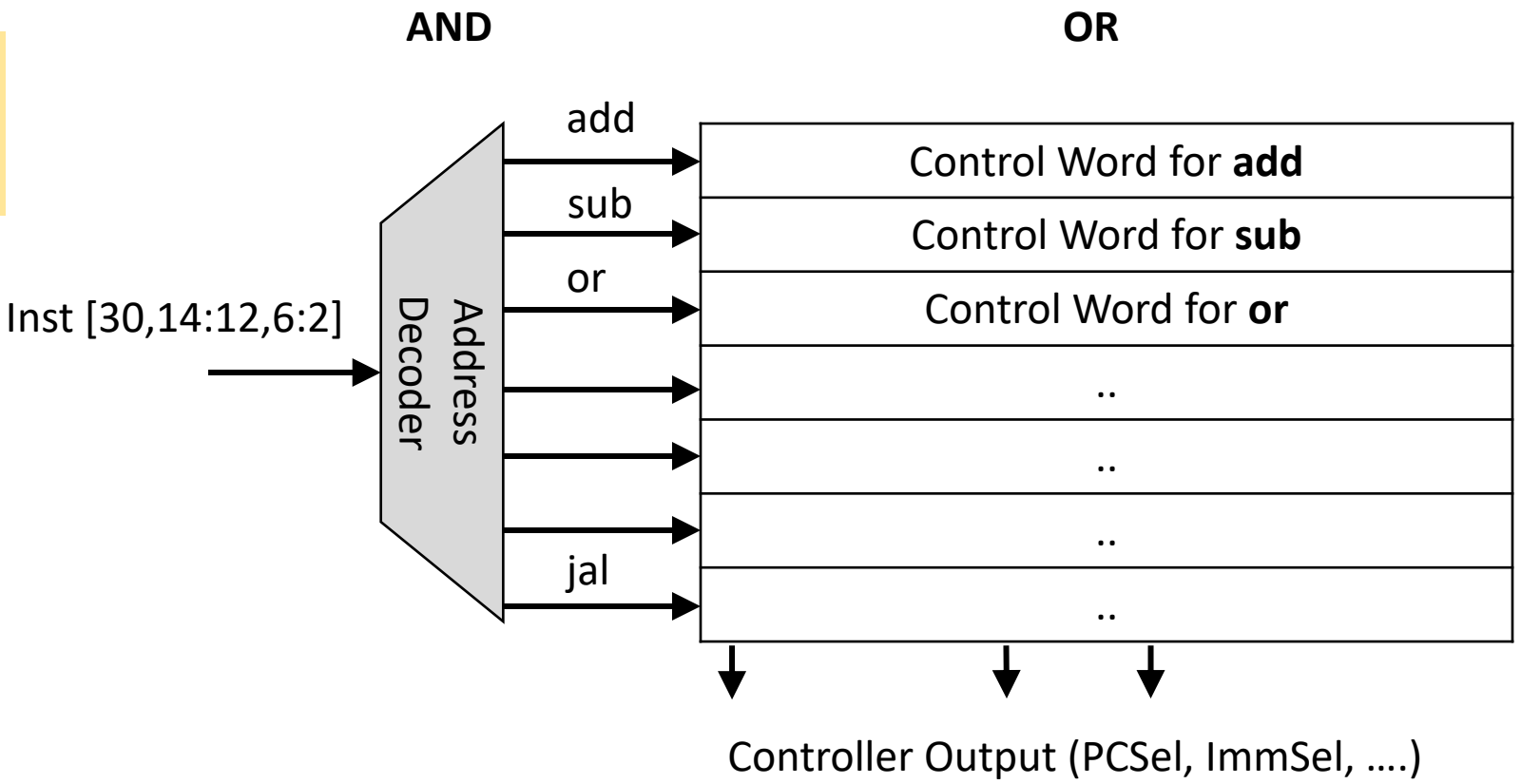
ROM-based Control

- Any logical transformation of N input bits to M output bits can be accomplished by a look-up table with 2^N entries (indexed by the input bits) of M bits each.
- look-up table can be simpler (less error-prone) and friendlier



ROM Controller Implementation

For each binary address input pattern, exactly one of the wordlines (horizontal) is activated by the address-decoder.



Control Logic to Decode add

inst[30] inst[14:12] inst[6:2]

0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
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

add = $\overline{i[30]}$ & $\overline{i[14]}$ & $\overline{i[13]}$ & $\overline{i[12]}$ & R-type

R-type = $\overline{i[6]}$ & i[5] & $\overline{i[4]}$ & $\overline{i[3]}$ & i[2] & RV32I

RV32I = i[1] & i[0]

Universität Bremen

RISC-V Datapath cont...

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