CS 211: Intro to Computer Architecture

12.2: Single-Cycle RV64I CPU – Mem, Branch, & Jump

Minesh Patel

Spring 2025 – Thursday 17 April

Announcements

- Ongoing
  - **WA8:** due **Friday (April 18) @ 11:59 pm**
  - **PA4:** due **Friday (April 18) @ 11:59 pm**

- Upcoming
  - **PA5:** TBA: planned for Friday
  - **WA9:** TBA: planned for Friday

Reference Material

- Today's lecture partially draws inspiration from:
  - CS 61C @ UC Berkeley (Prof. Dan Garcia)

And the RISC-V ISA Manual

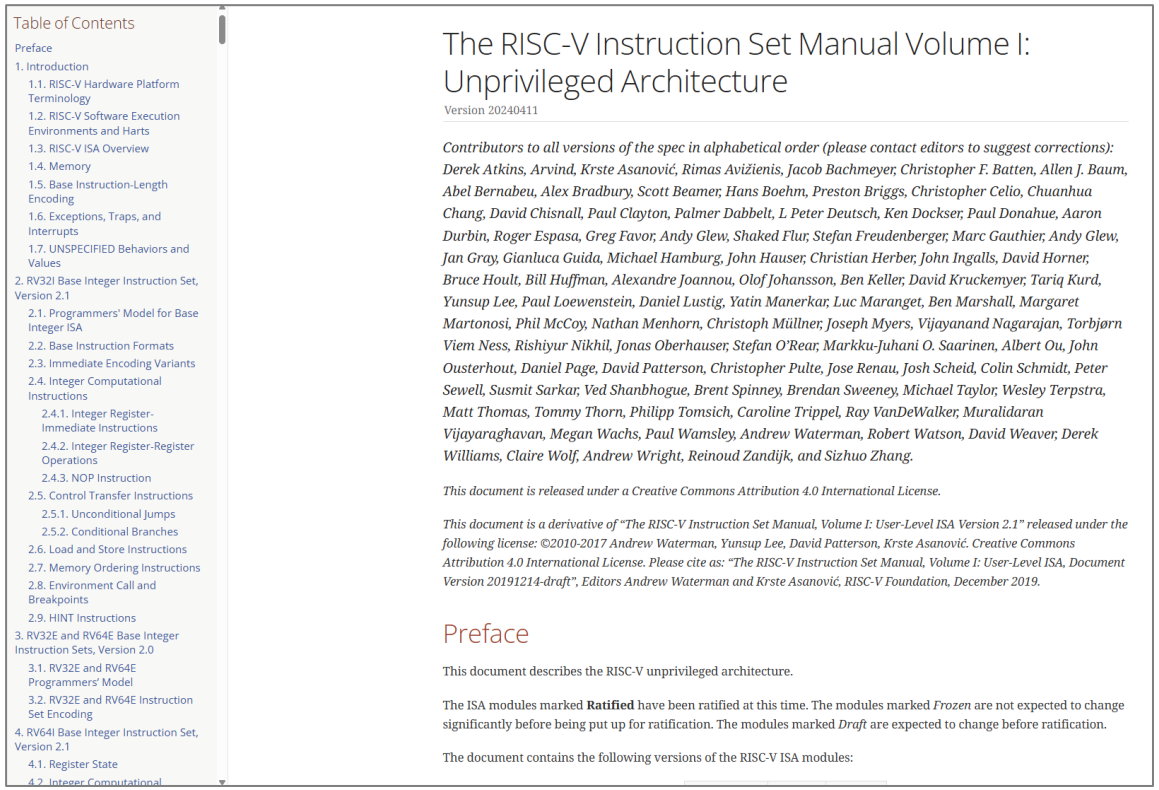


Table of Contents

The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture

Version 20240411

Contributors to all versions of the spec in alphabetical order (please contact editors to suggest corrections): Derek Atkins, Arvind, Krste Asanović, Rimas Avižienis, Jacob Bachmeyer, Christopher F. Batten, Allen J. Baum, Abel Bernabeu, Alex Bradbury, Scott Beamer, Hans Boehm, Preston Briggs, Christopher Celio, Chuanhua Chang, David Chisnall, Paul Clayton, Palmer Dabbelt, L Peter Deutsch, Ken Dockser, Paul Donahue, Aaron Durbin, Roger Espasa, Greg Favor, Andy Glew, Shaked Flur, Stefan Freudenberger, Marc Gauthier, Andy Glew, Jan Gray, Gianluca Guida, Michael Hamburg, John Hauser, Christian Herber, John Ingalls, David Horner, Bruce Hoult, Bill Huffman, Alexandre Joannou, Olof Johansson, Ben Keller, David Kruckemyer, Tariq Kurd, Yunsup Lee, Paul Loewenstein, Daniel Lustig, Yatin Manerkar, Luc Maranget, Ben Marshall, Margaret Martonosi, Phil McCoy, Nathan Menhorn, Christoph Müllner, Joseph Myers, Vijayanand Nagarajan, Torbjørn Viem Ness, Rishiyur Nikhil, Jonas Oberhauser, Stefan O'Rear, Markku-Juhani O. Saarinen, Albert Ou, John Ousterhout, Daniel Page, David Patterson, Christopher Pulte, Jose Renau, Josh Scheid, Colin Schmidt, Peter Sewell, Susmit Sarkar, Ved Shanbhogue, Brent Spinney, Brendan Sweeney, Michael Taylor, Wesley Terpstra, Matt Thomas, Tommy Thorn, Philipp Tomsich, Caroline Trippel, Ray VanDeWalker, Muralidaran Vijayaraghavan, Megan Wachs, Paul Wamsley, Andrew Waterman, Robert Watson, David Weaver, Derek Williams, Claire Wolf, Andrew Wright, Reinoud Zandijk, and Sizhuo Zhang.

Preface

This document describes the RISC-V unprivileged architecture.

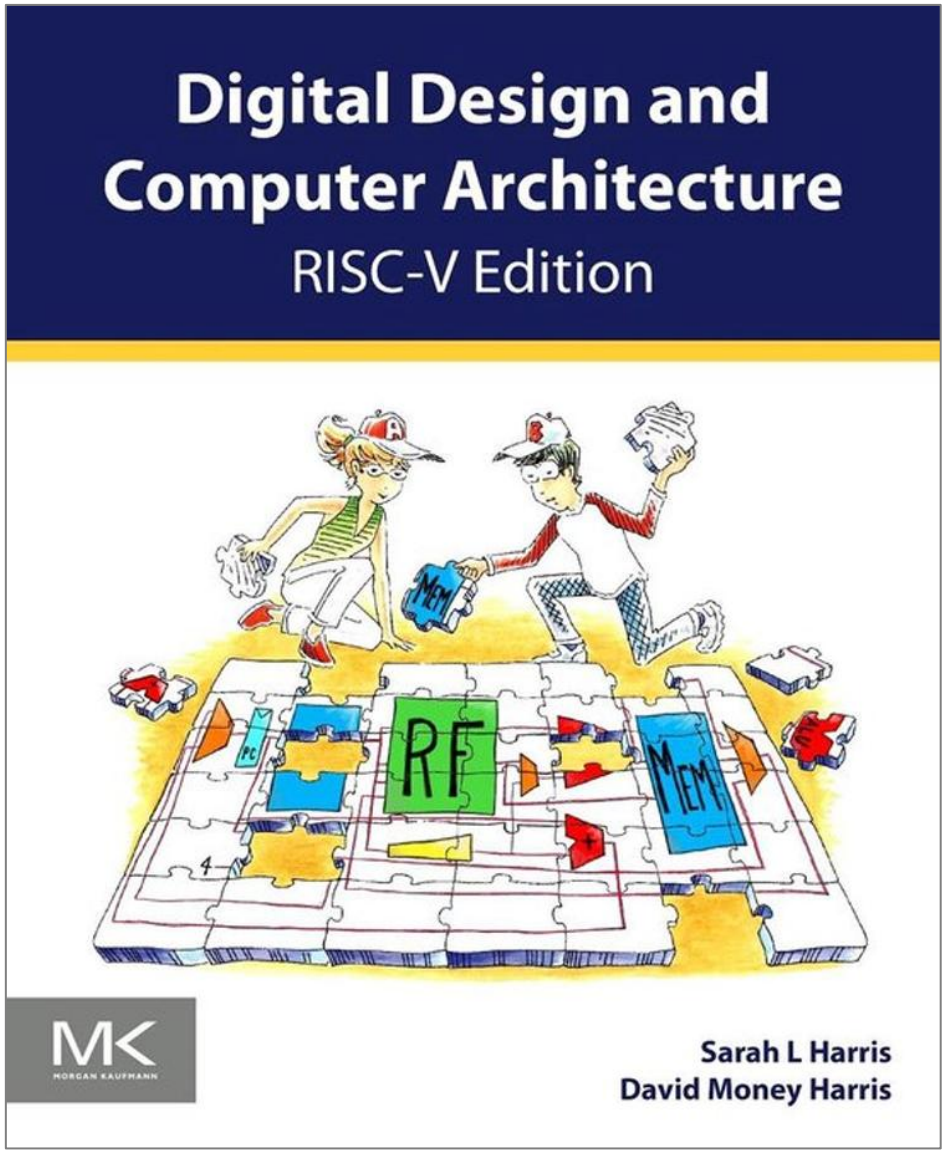
The ISA modules marked **Ratified** have been ratified at this time. The modules marked *Frozen* are not expected to change significantly before being put up for ratification. The modules marked *Draft* are expected to change before ratification.

The document contains the following versions of the RISC-V ISA modules:

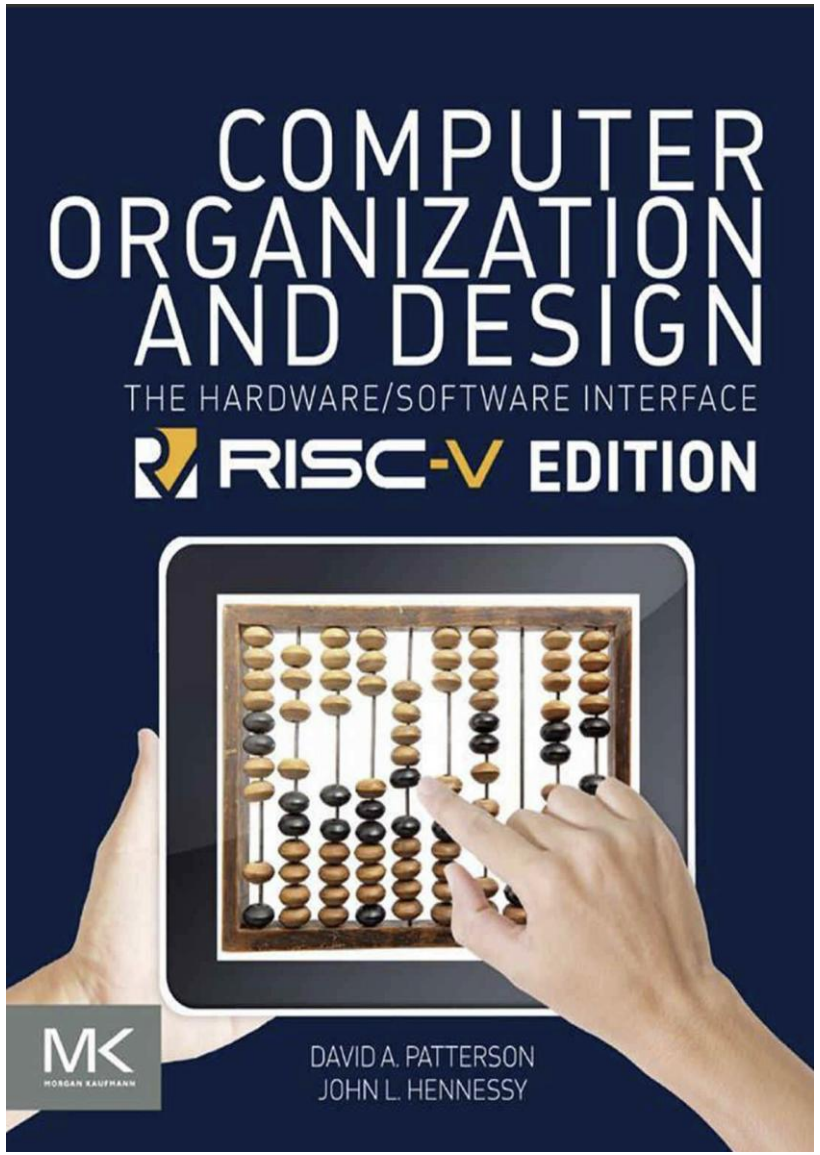
RISC-V ISA Manual (Online Version)

Recommended Reading

Section 7.3: "Single-Cycle Processor"

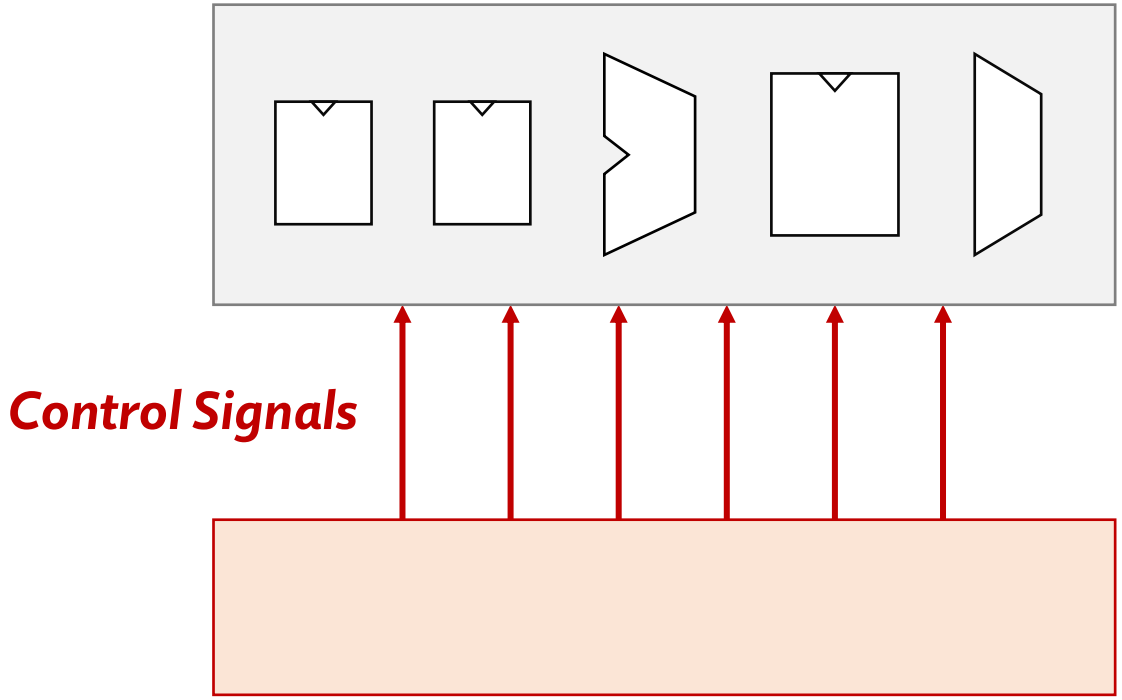


Section 4.3: "Building a Datapath"



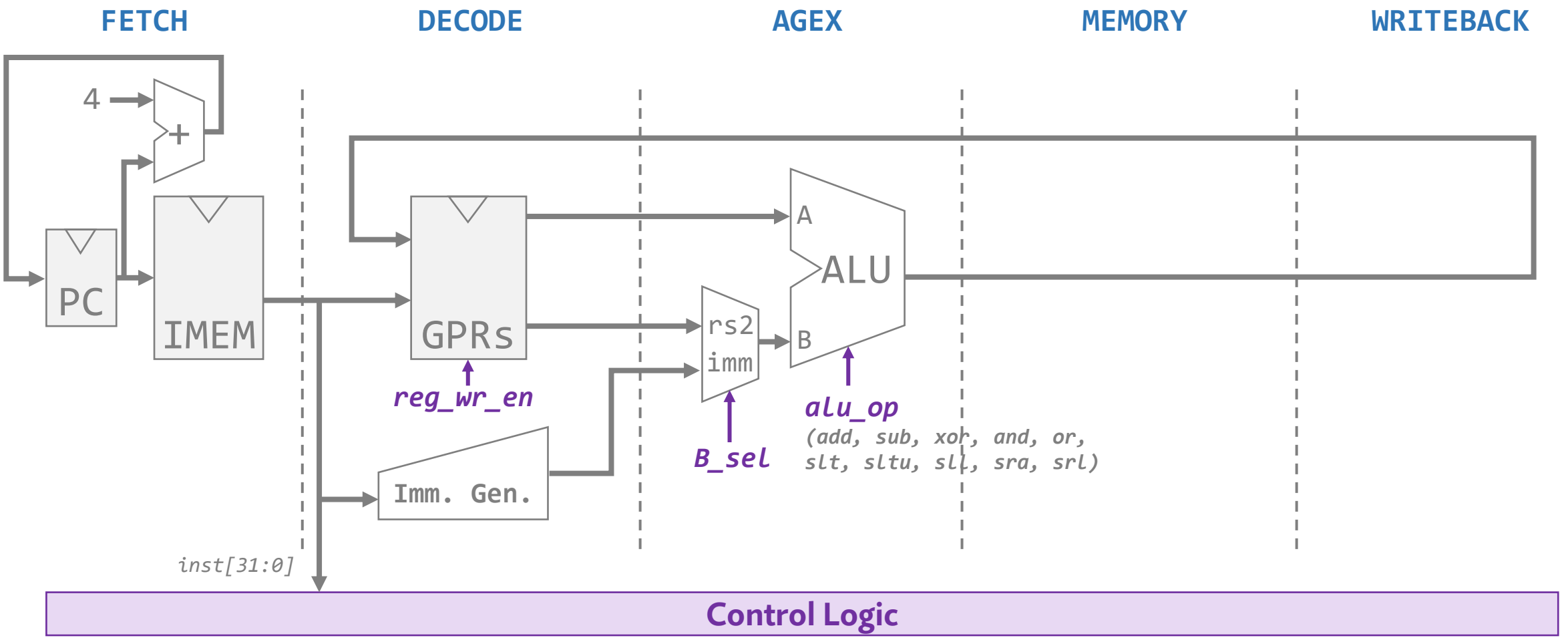
Summary: Overall Design

Datapath: all logic elements we need



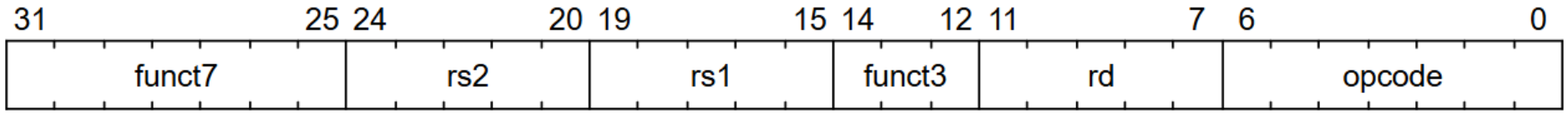
Control Logic: orchestrates the datapath

Our CPU So Far: Arithmetic/Logic



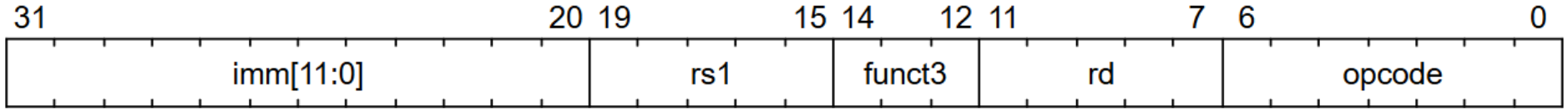
Today's goal: implement everything else!

6 Types of RISC-V Instructions



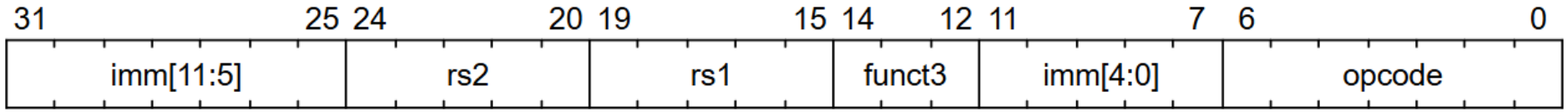
R-Type

add, sub, slt(u),
xor, or, and,
sll, srl, sra



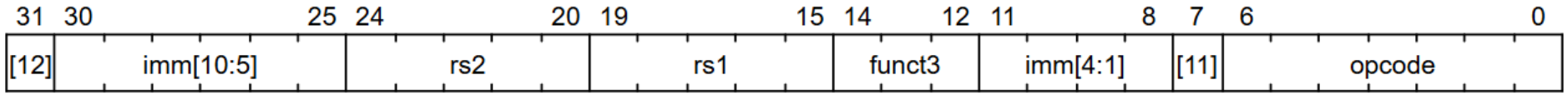
I-Type

addi, subi, slt(u)i,
xori, ori, andi, slli,
srli, srai, lb(u),
lh(u), lw(u), ld, jalr



S-Type

sb, sh, sw, sd

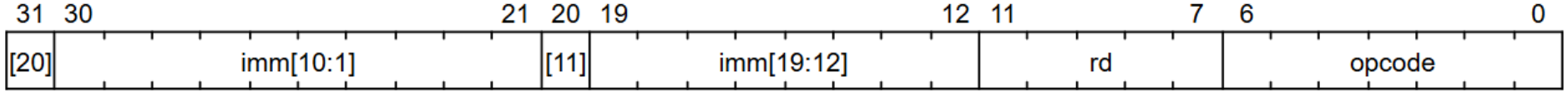


B-Type

b<cond>

U-Type

lui, auipc



J-Type

jal

Agenda

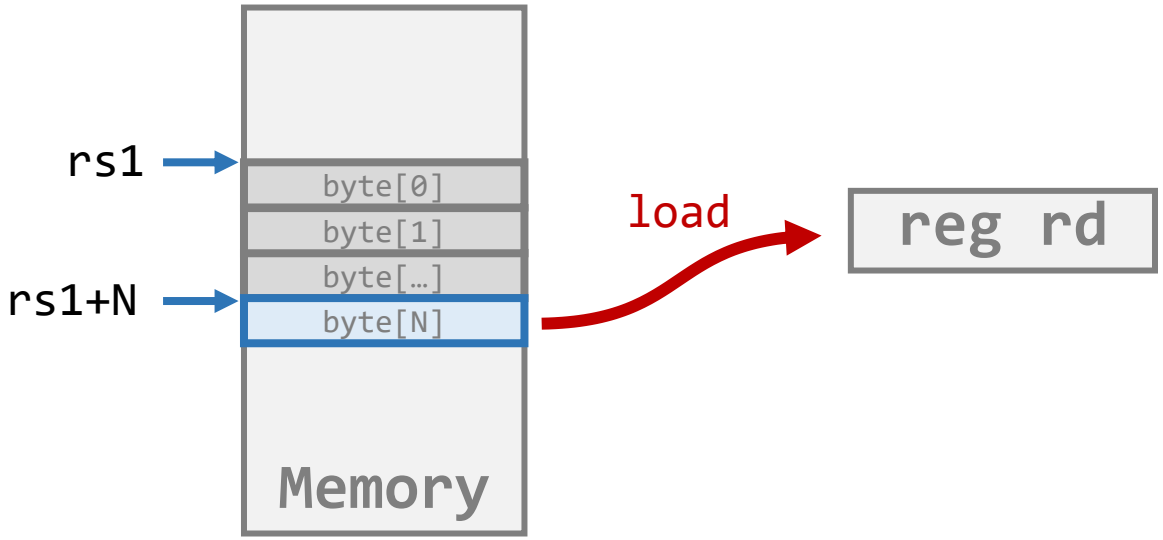
- **Memory Operations**
  - I-Type: Loads
  - S-Type: Stores
- Control Flow Operations
  - B-Type: Branches
  - Jumps and J-Type
  - U-Type: AUIPC and LUI

RV64I Memory Operations

Load

```
lb rd, N(rs1)
```

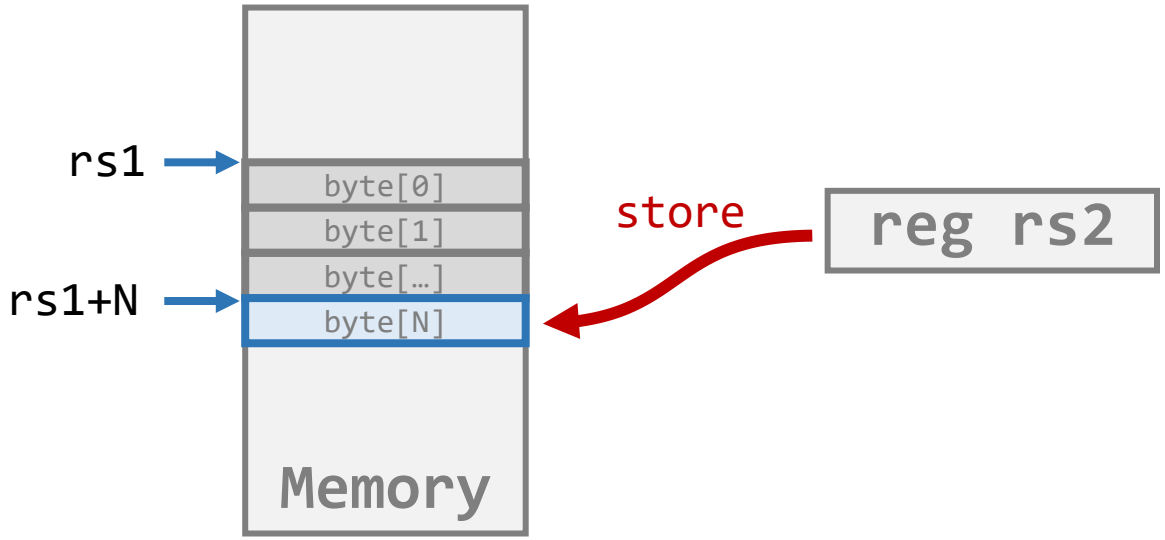
```
rd = mem[rs1 + N]
```



Store

```
sb rs2, N(rs1)
```

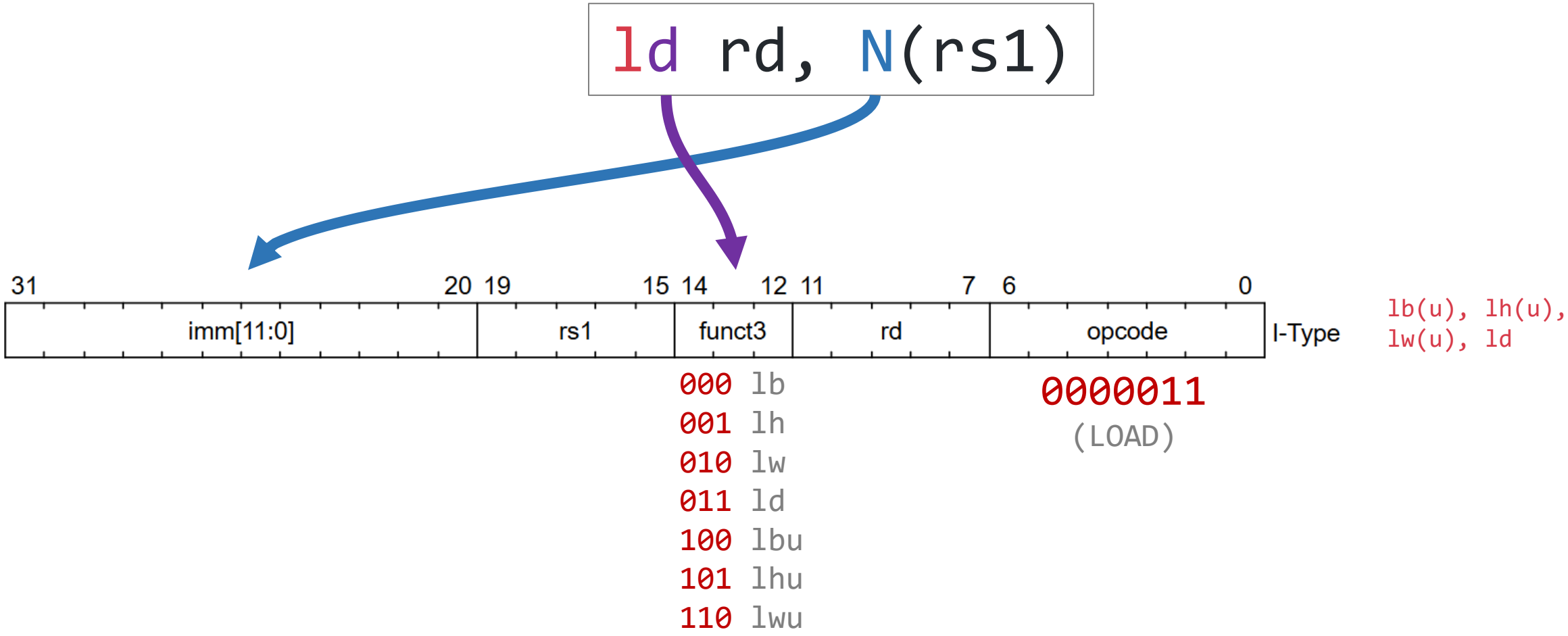
```
mem[rs2 + N] = rs1
```



Agenda

- Memory Operations
  - **I-Type: Loads**
  - S-Type: Stores
- Control Flow Operations
  - B-Type: Branches
  - Jumps and J-Type
  - U-Type: AUIPC and LUI

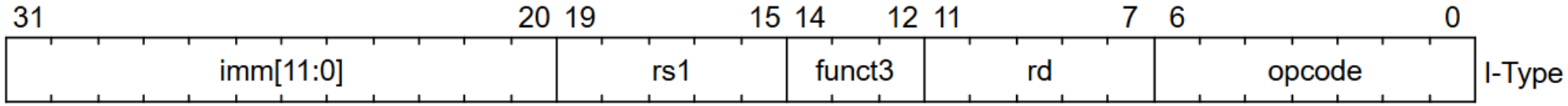
LOAD Instructions (I-Type)



- Same encoding as the **I-Type instructions** we already implemented

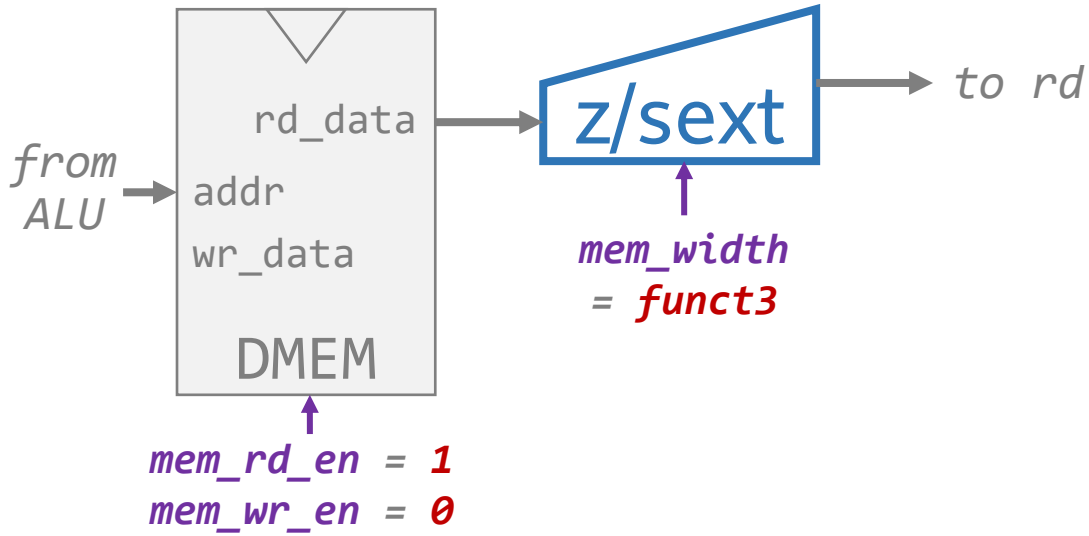
Implementing LOAD Variants

```
ld/w/h/b(u) rd, N(rs1)
```




```
000 lb
001 lh
010 lw
011 ld
100 lbu
101 lhu
110 lwu
```

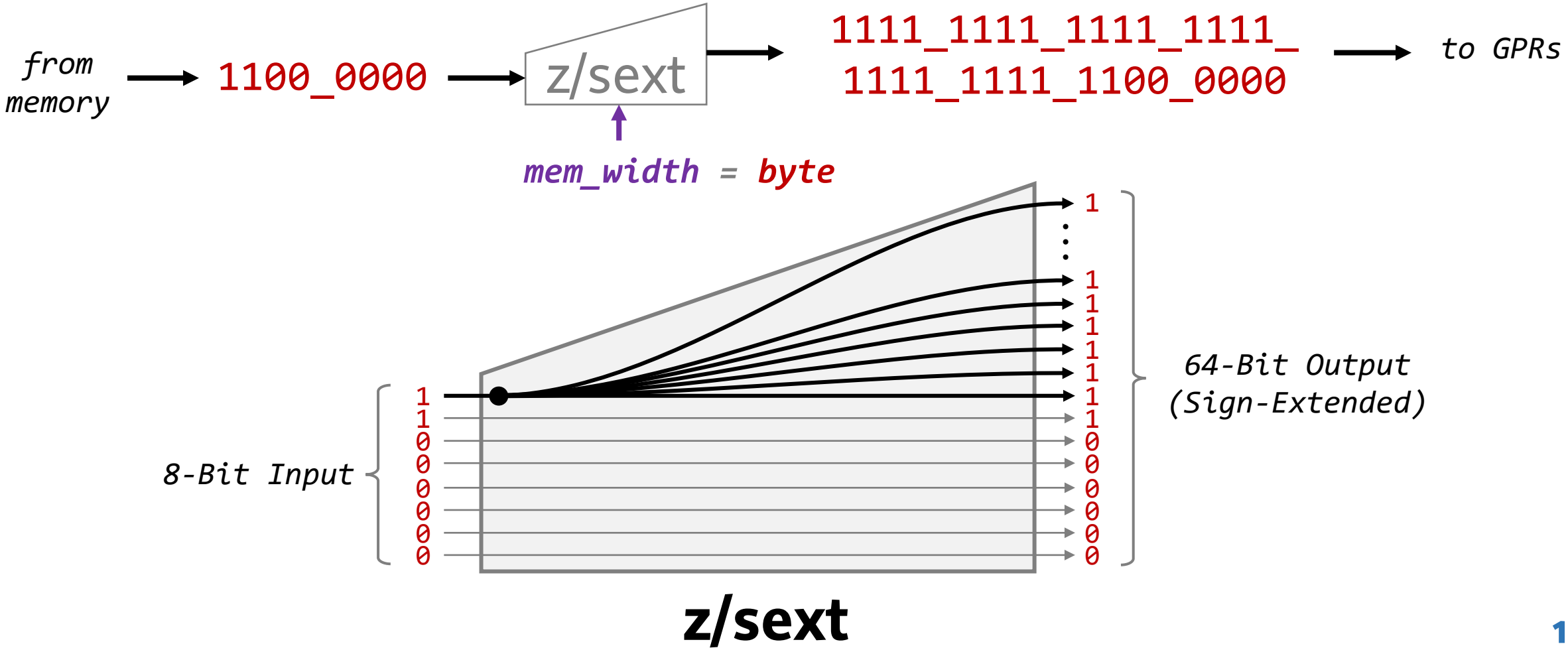
```
0000011
(LOAD)
```



- **New logic block:** zero and sign extension
- **New control signal:** `mem_width`

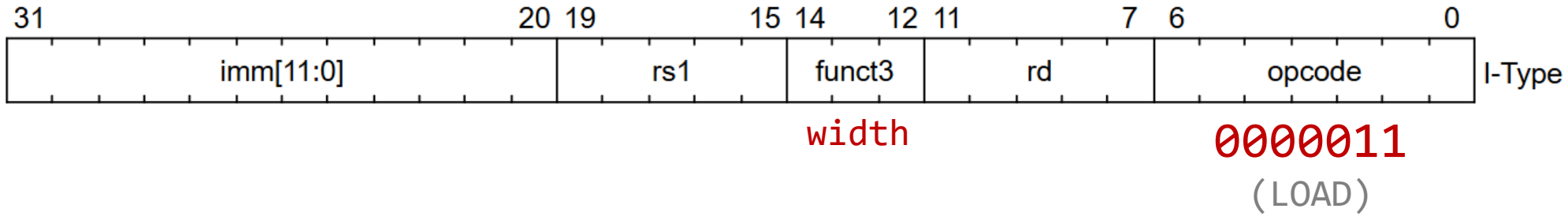
Example Sign Extension: lb

- Easy to implement (just a couple gates to choose the width)



LOAD: State Updates

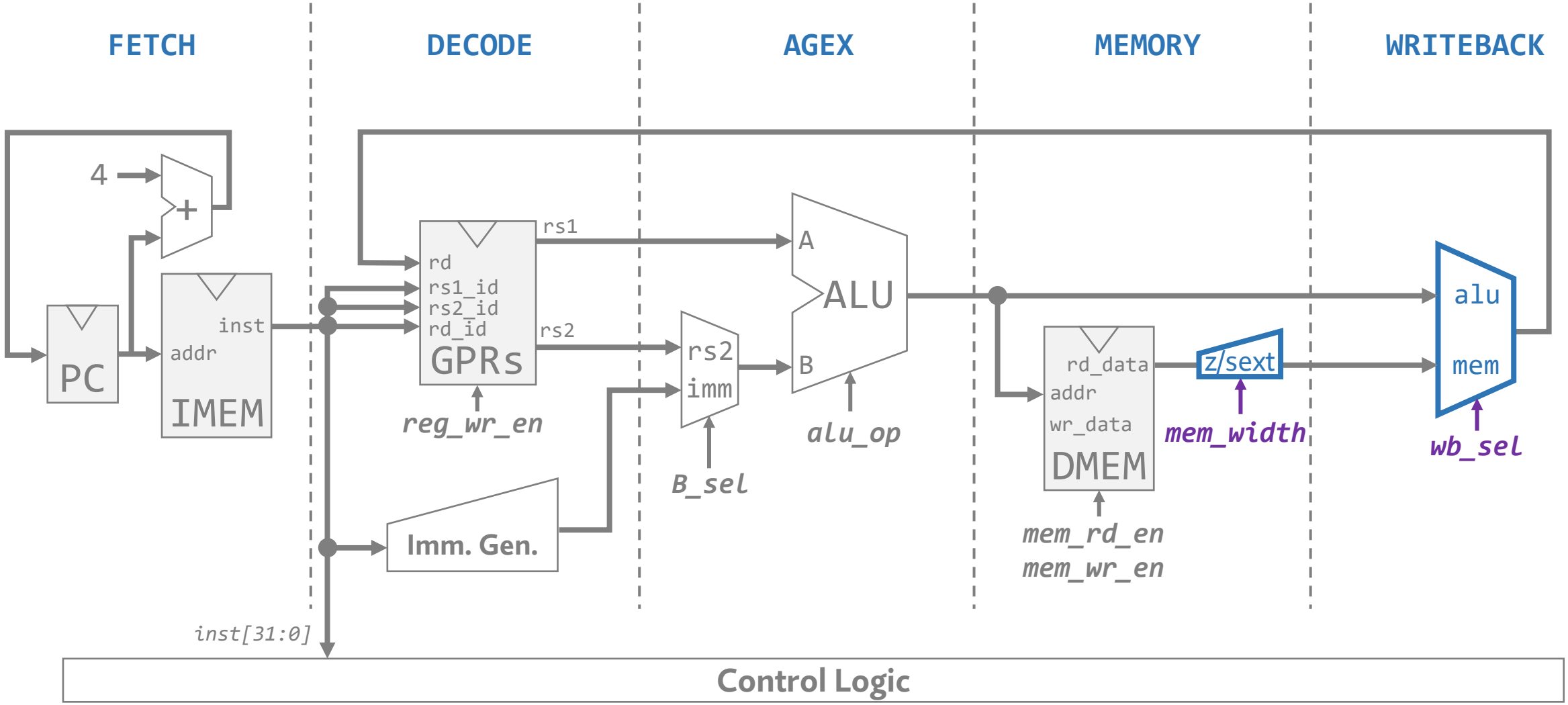
```
ld/w/h/b(u) rd, N(rs1)
```



State Element	Action
PC	PC += 4
GPRs	WRITE(R[rd]) and READ(R[rs1])
DMEM	READ: MEM[R[rs1] + sext(imm[11:0])]

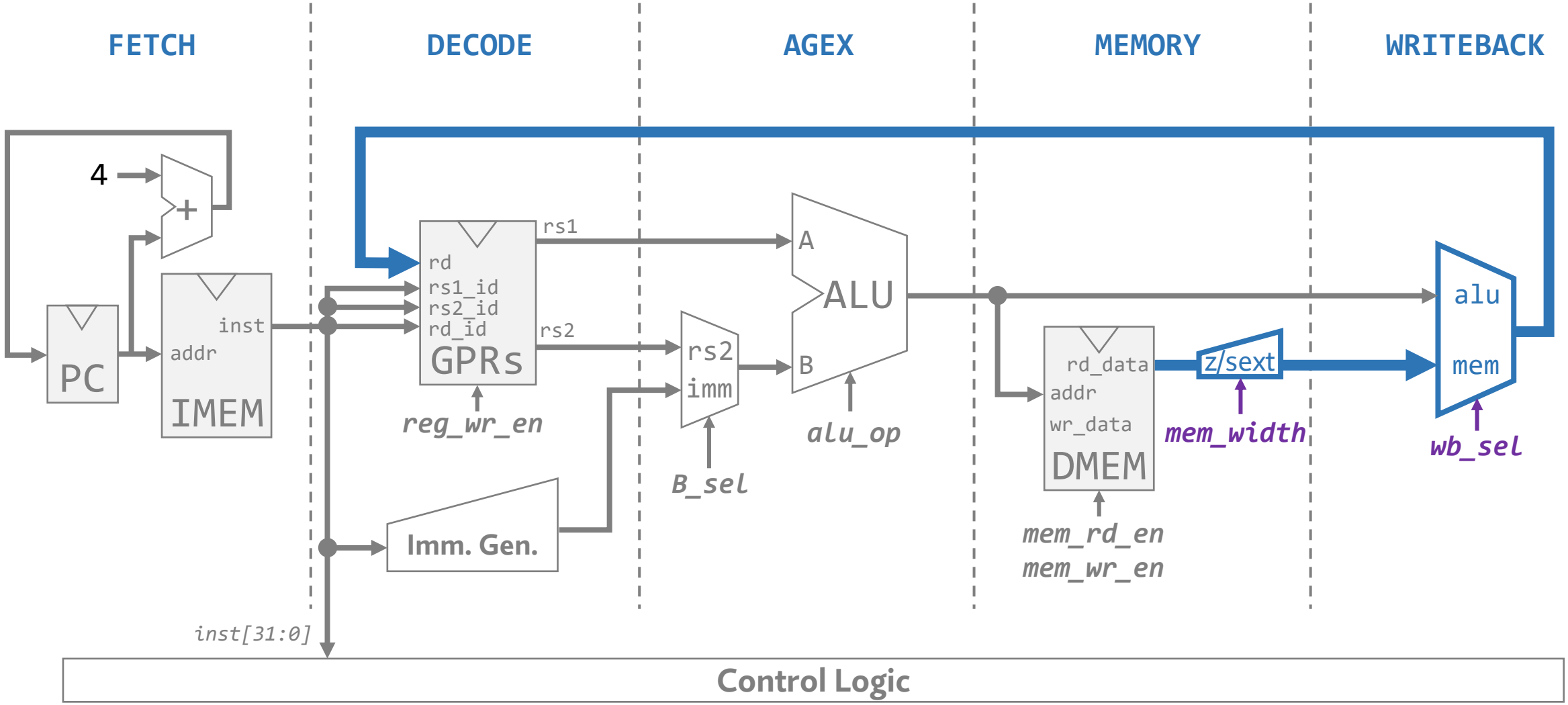
New Components

State Element	Action
PC	PC += 4
GPRs	WRITE(R[rd]) and READ(R[rs1])
DMEM	READ: MEM[R[rs1] + sext(imm[11:0])]



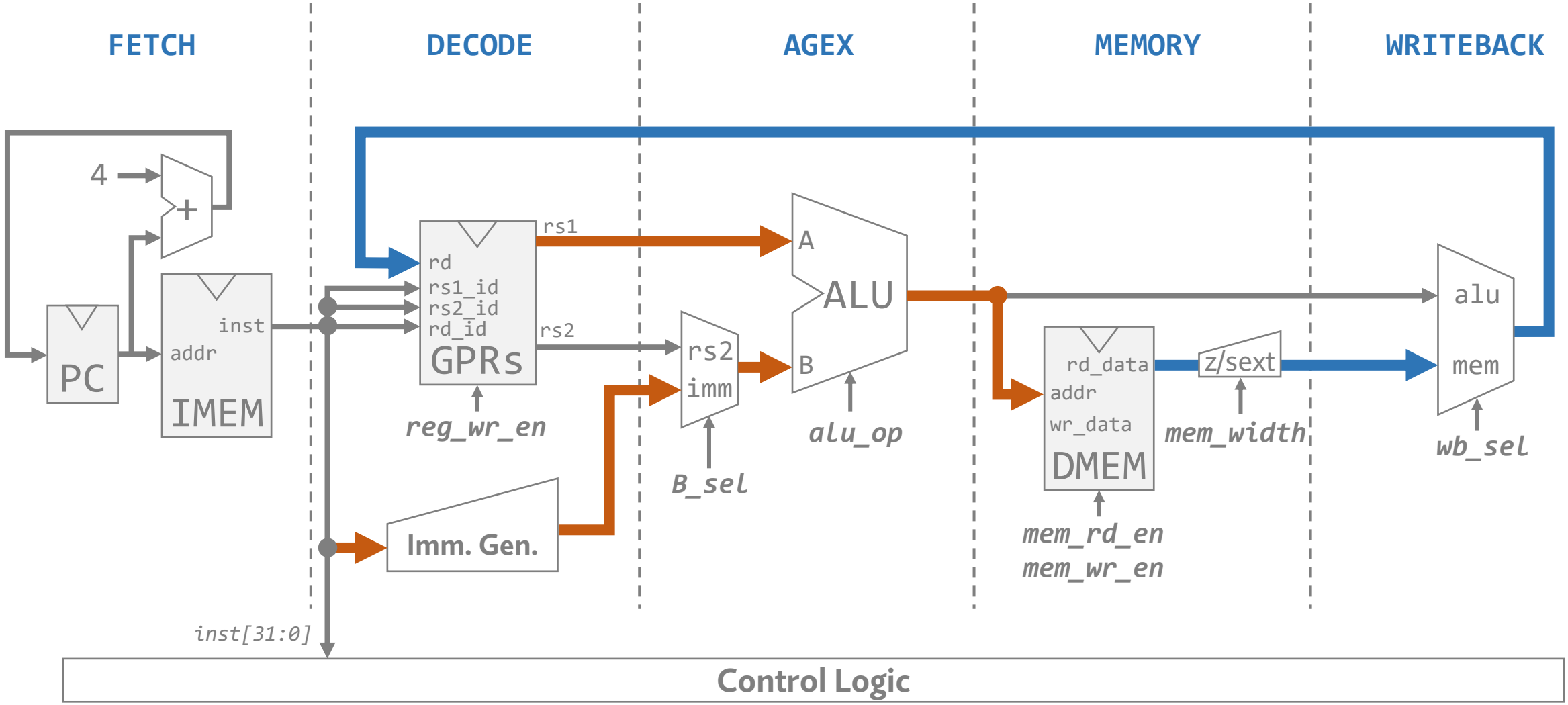
Memory to Registers

State Element	Action
PC	PC += 4
GPRs	WRITE(R[rd]) and READ(R[rs1])
DMEM	READ: MEM[R[rs1] + sext(imm[11:0])]



Datapath for LOAD

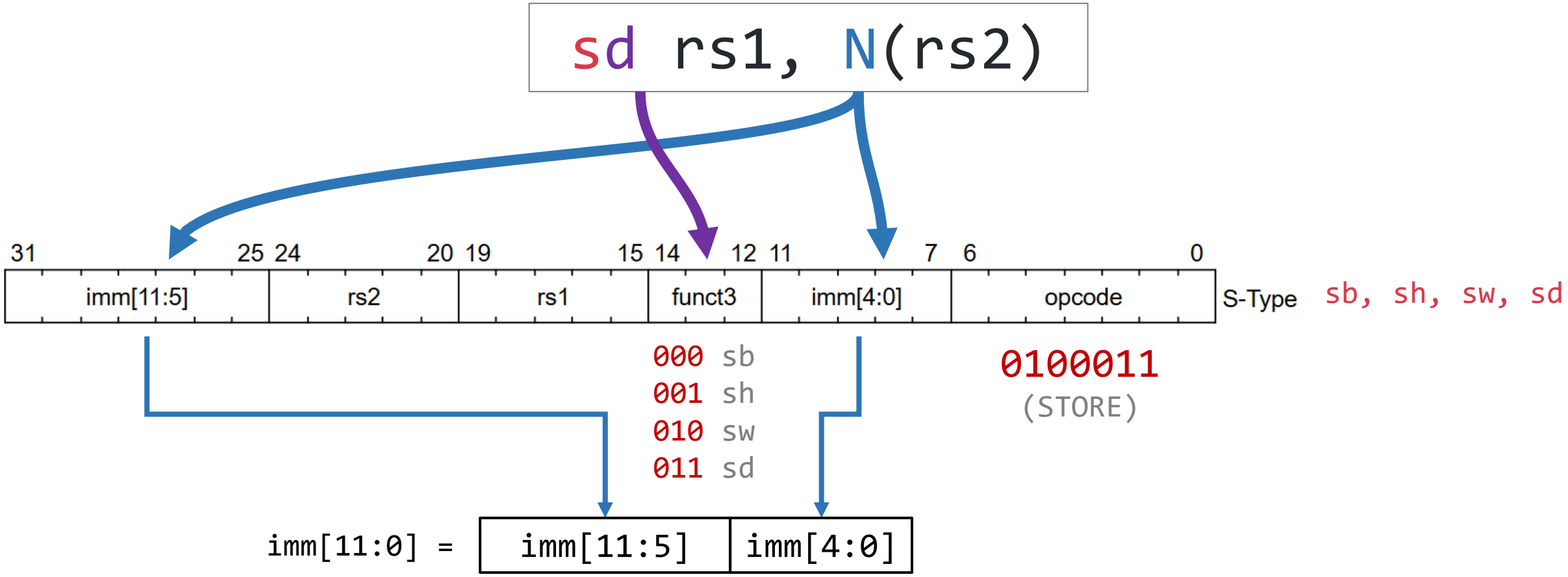
State Element	Action
PC	PC += 4
GPRs	WRITE(R[rd]) and READ(R[rs1])
DMEM	READ: MEM[R[rs1] + sext(imm[11:0])]



Agenda

- Memory Operations
  - I-Type: Loads
  - **S-Type: Stores**
- Control Flow Operations
  - B-Type: Branches
  - Jumps and J-Type
  - U-Type: AUIPC and LUI

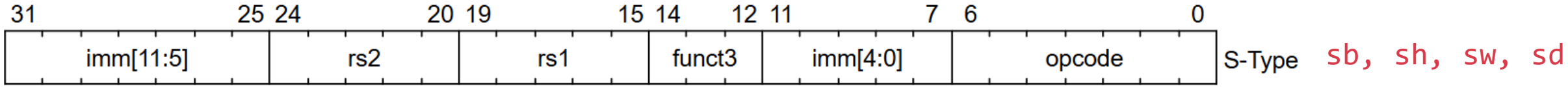
STORE Instructions (S-Type)



- **Key idea:** `rs2` bits are always in the same place (easy to decode)
  - **Consequence:** fragmented `imm` field

STORE Instructions (S-Type)

```
sd/w/h/b rs2, N(rs1)
```



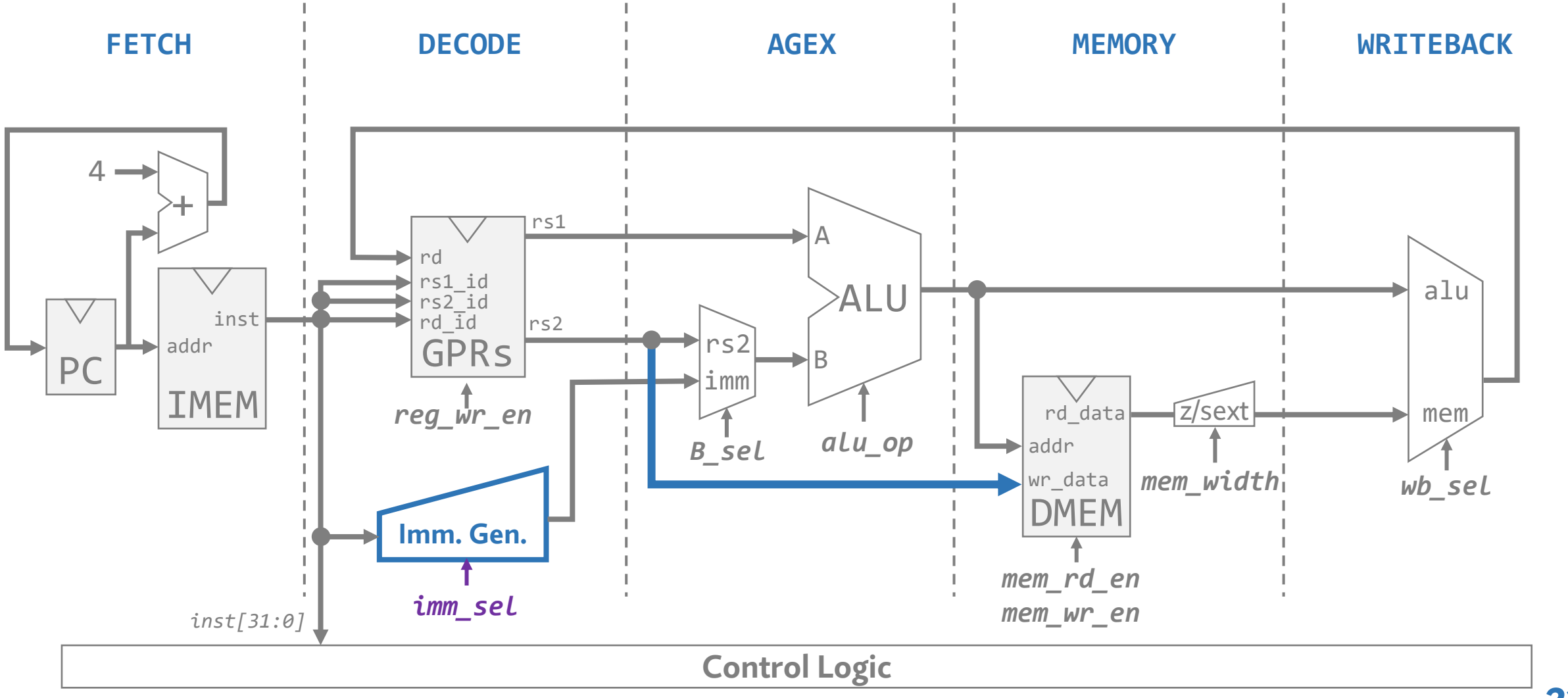
000 sb
001 sh
010 sw
011 sd

0100011
(STORE)

State Element	Action
PC	PC += 4
GPRs	READ(R[rs1] & R[rs2])
DMEM	WRITE: MEM[R[rs1] + sext(imm[11:0])]

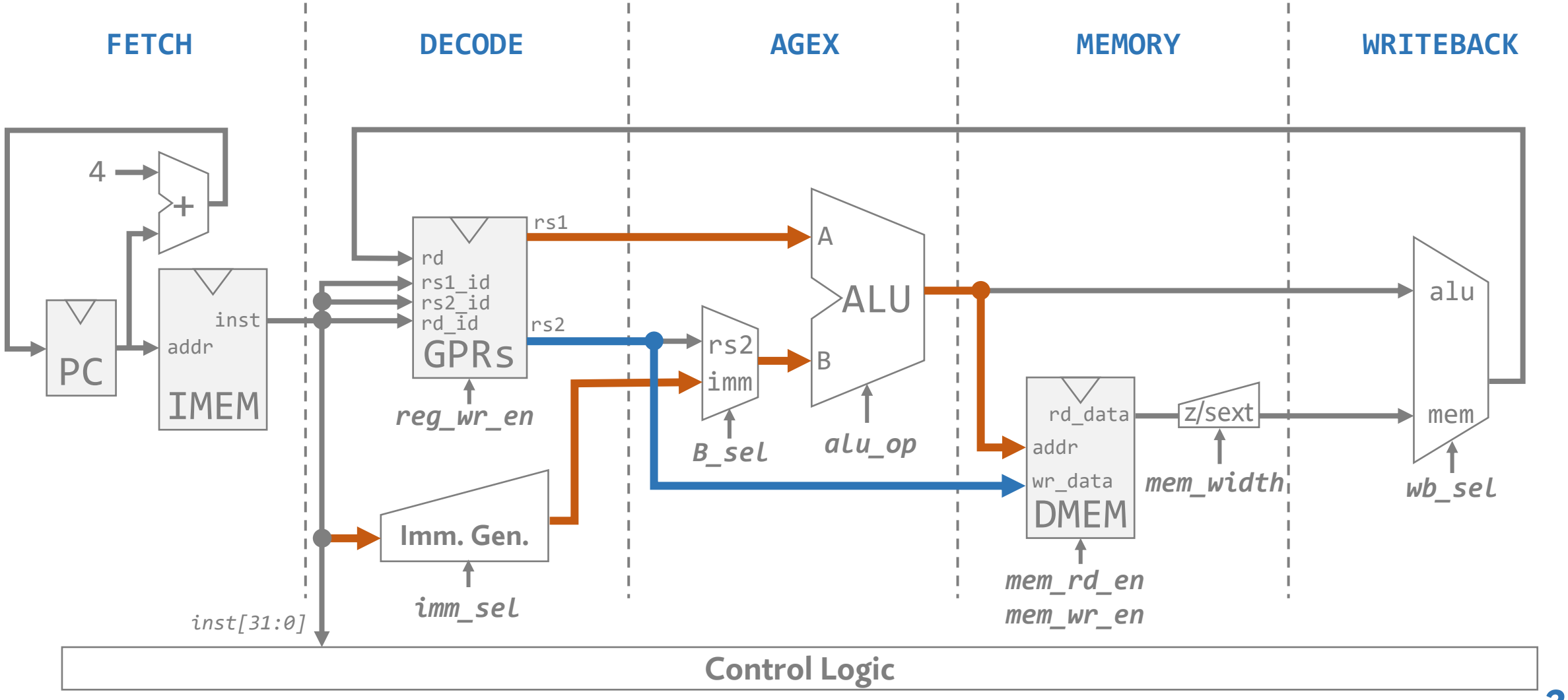
Updated Components

State Element	Action
PC	PC += 4
GPRs	READ(R[rs1] & R[rs2])
DMEM	WRITE: MEM[R[rs1] + sext(imm[11:0])]



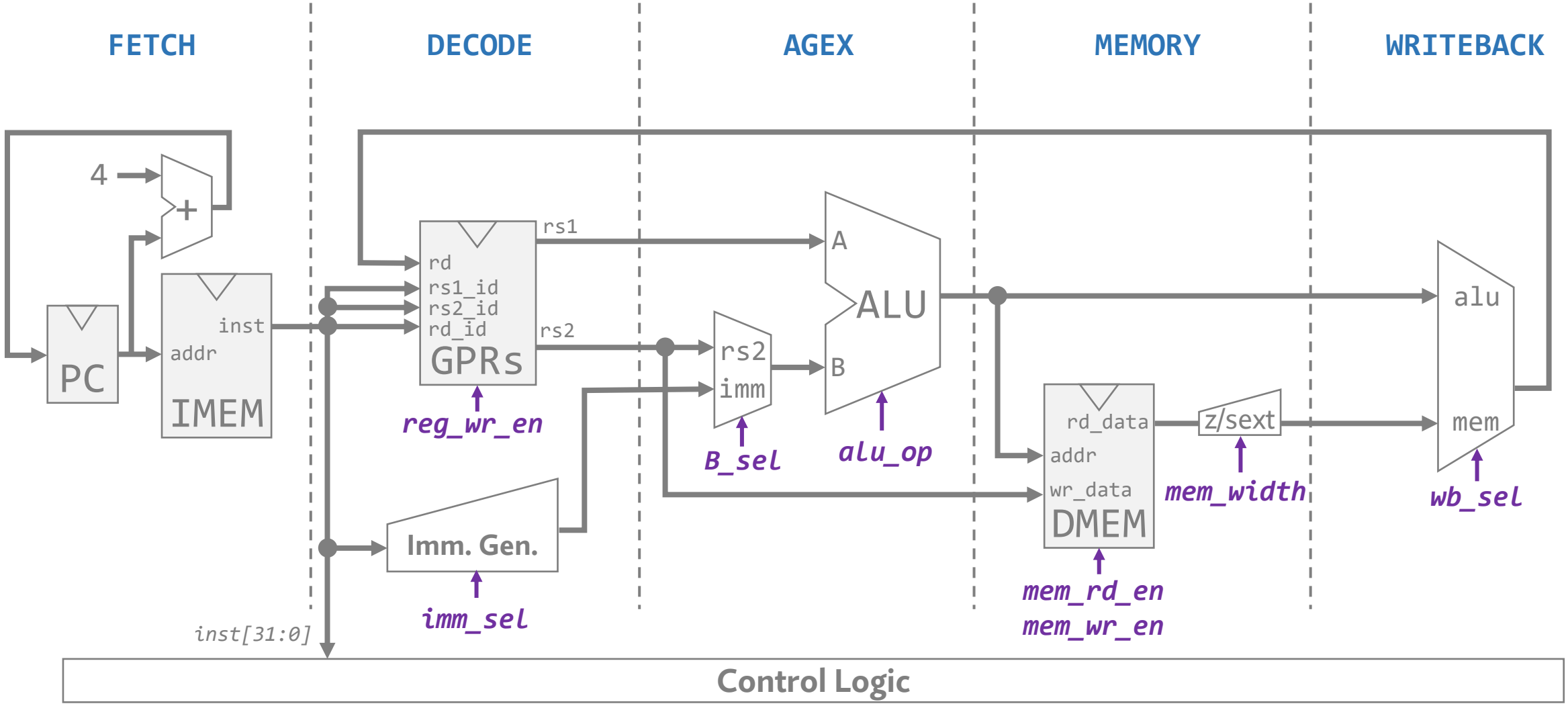
Datapath for STORE

State Element	Action
PC	PC += 4
GPRs	READ(R[rs1] & R[rs2])
DMEM	WRITE: MEM[R[rs1] + sext(imm[11:0])]



Our CPU So Far

- **Arith/logic:** add(i), sub, slt(u)(i), xor(i), or(i), and(i), sll(i), srl(i), sra(i)
- **Loads:** lb(u), lh(u), lw(u), ld,
- **Stores:** sb, sh, sw, sd



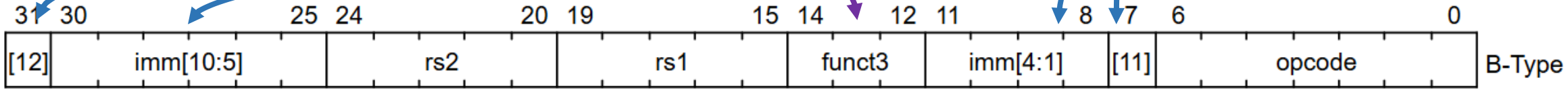
Agenda

- Memory Operations
  - I-Type: Loads
  - S-Type: Stores

- **Control Flow Operations**
  - **B-Type: Branches**
  - Jumps and J-Type
  - U-Type: AUIPC and LUI

Branch Instructions (B-Type)

```
b<cond> rs1, rs2, <PC offset>
```




```
000 eq
001 ne
100 lt
101 ge
110 ltu
111 geu
```

1100011
(BRANCH)

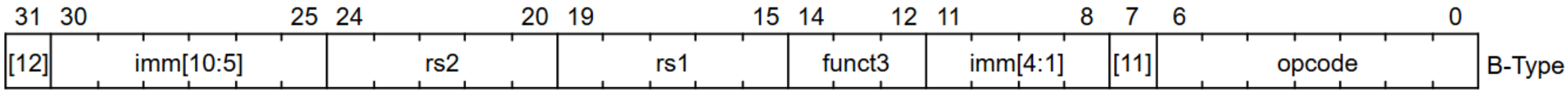
imm[12:0] =	inst[31]	inst[7]	inst[30:25]	inst[11:8]	0

five-piece immediate!

- **{rs2, rs1, funct3, opcode}** are in the same places as R/I/S-Type

Branch Instructions (B-Type)

```
b<cond> rs1, rs2, <PC offset>
```




```
000 eq
001 ne
100 lt
101 ge
110 ltu
111 geu
```

```
1100011
(BRANCH)
```

First time we're modifying the PC!

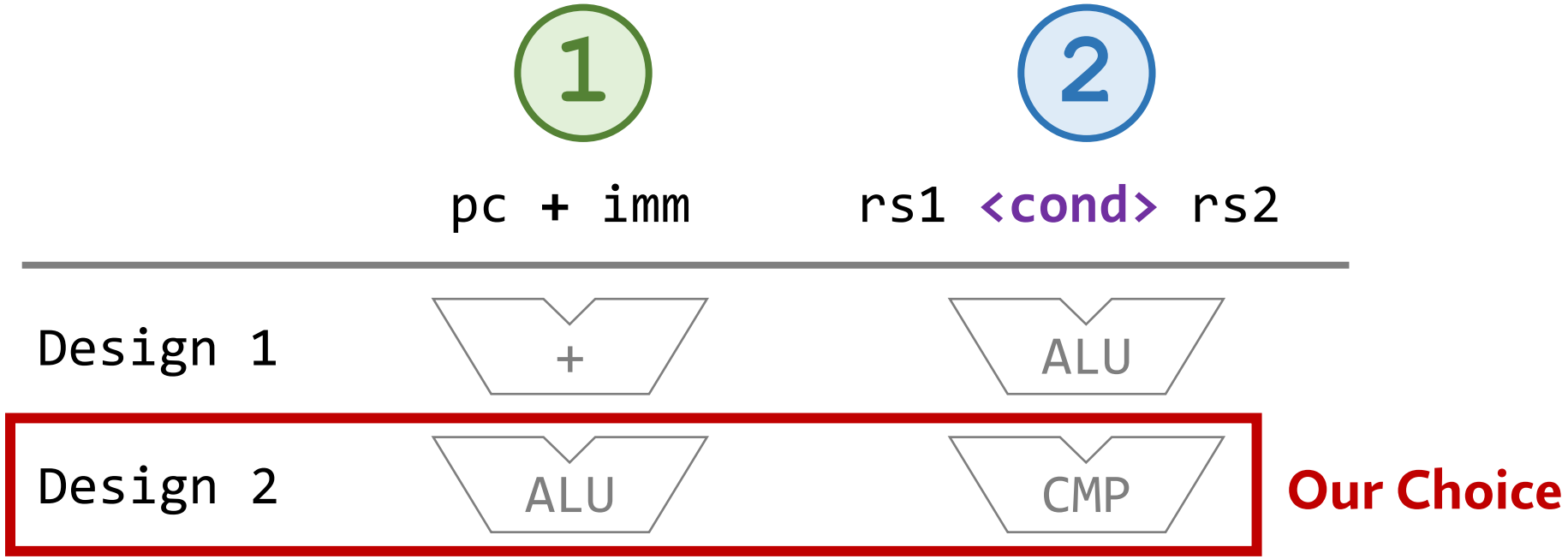
State Element	Action
PC	`PC += taken ? sext(imm[12:0]) : 4;`
GPRs	`READ(R[rs1] & R[rs2])`
DMEM	-

Evaluating the Branch Condition

```
b<cond> rs1, rs2, <PC offset>
```

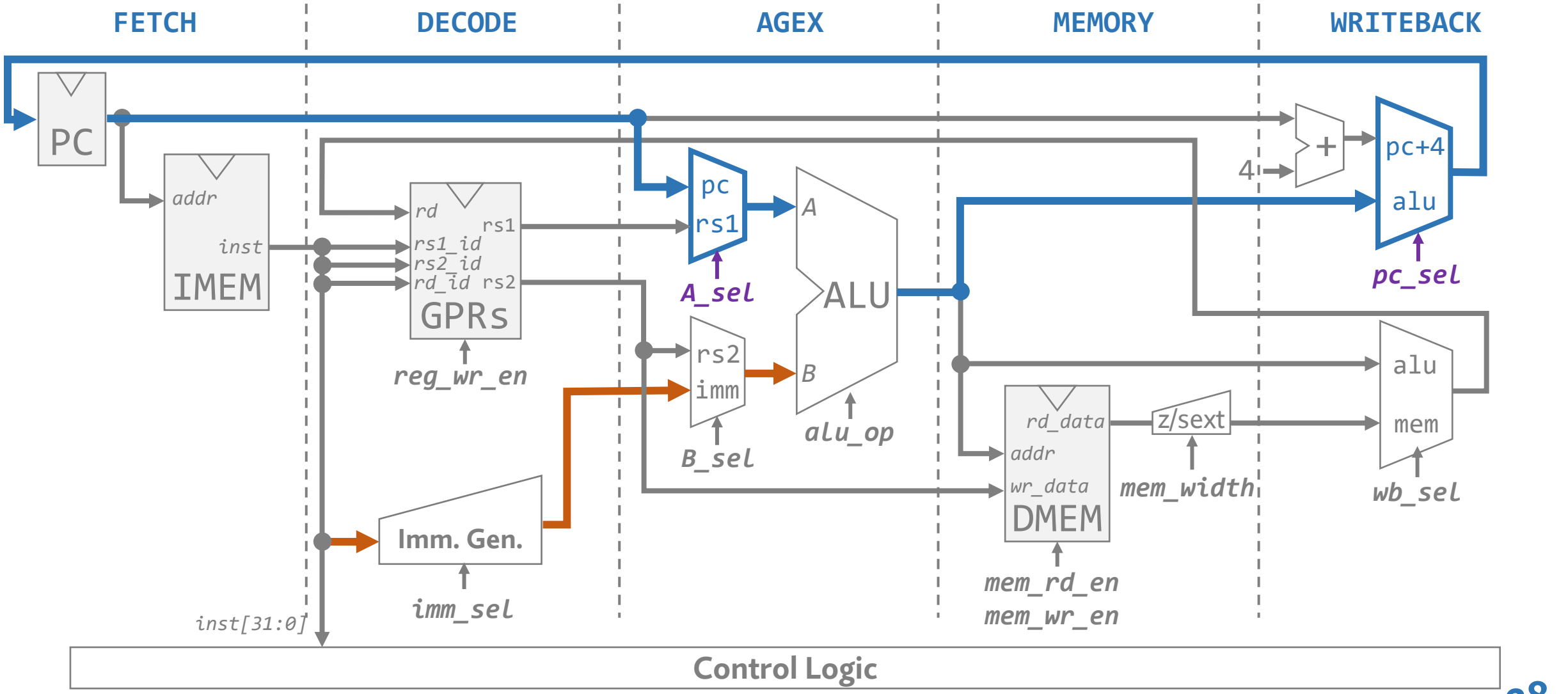
```
<cond>
000 eq
001 ne
100 lt
101 ge
110 ltu
111 geu
```

- We have **two separate calculations** to do
- We only have **one ALU**



1. Use ALU for pc+imm

State Element	Action
PC	PC += taken ? sext(imm[12:0]) : 4
GPRs	READ(R[rs1] & R[rs2])
DMEM	-

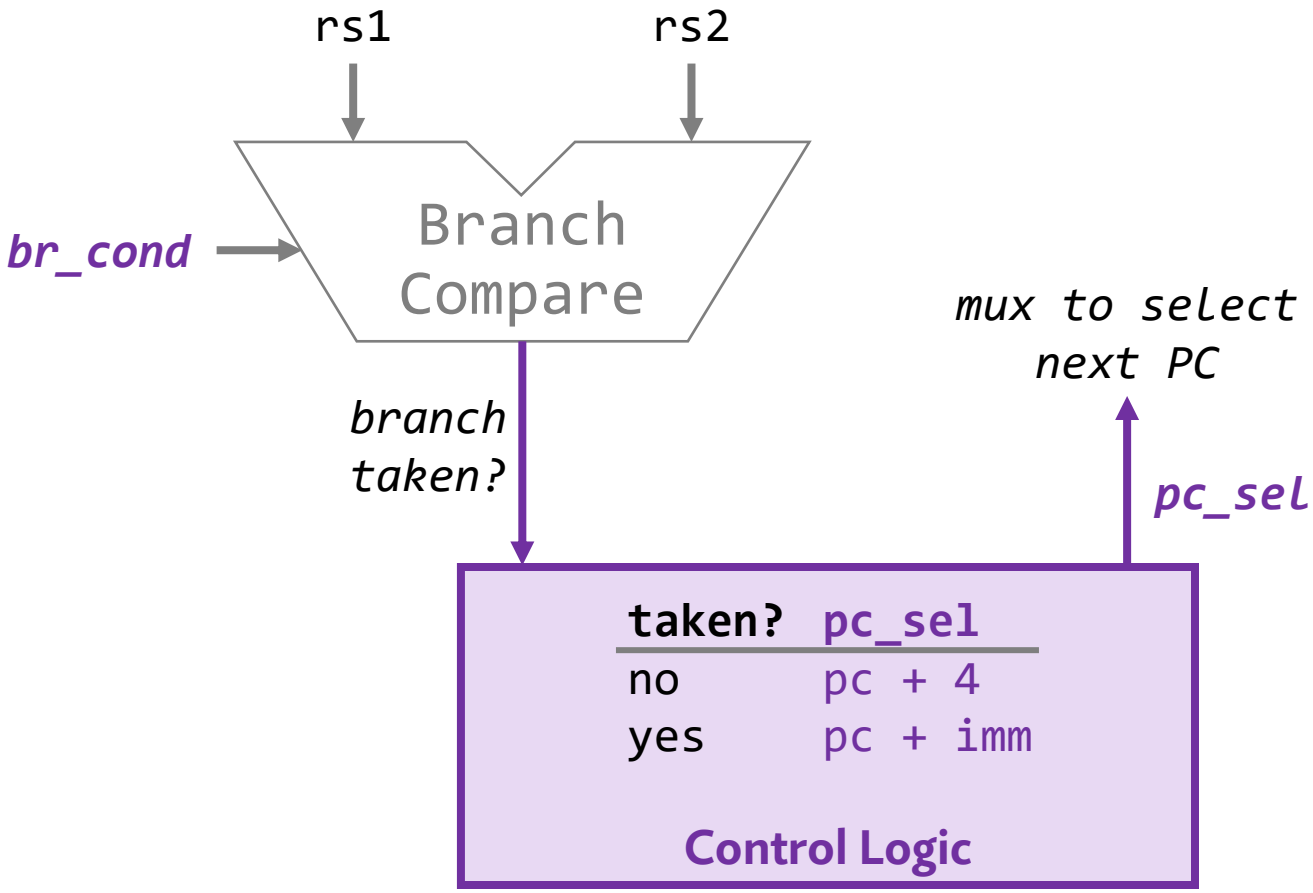


Branch Comparator

- Decides whether a **branch is taken (or not)**
- Just another combinational logic block

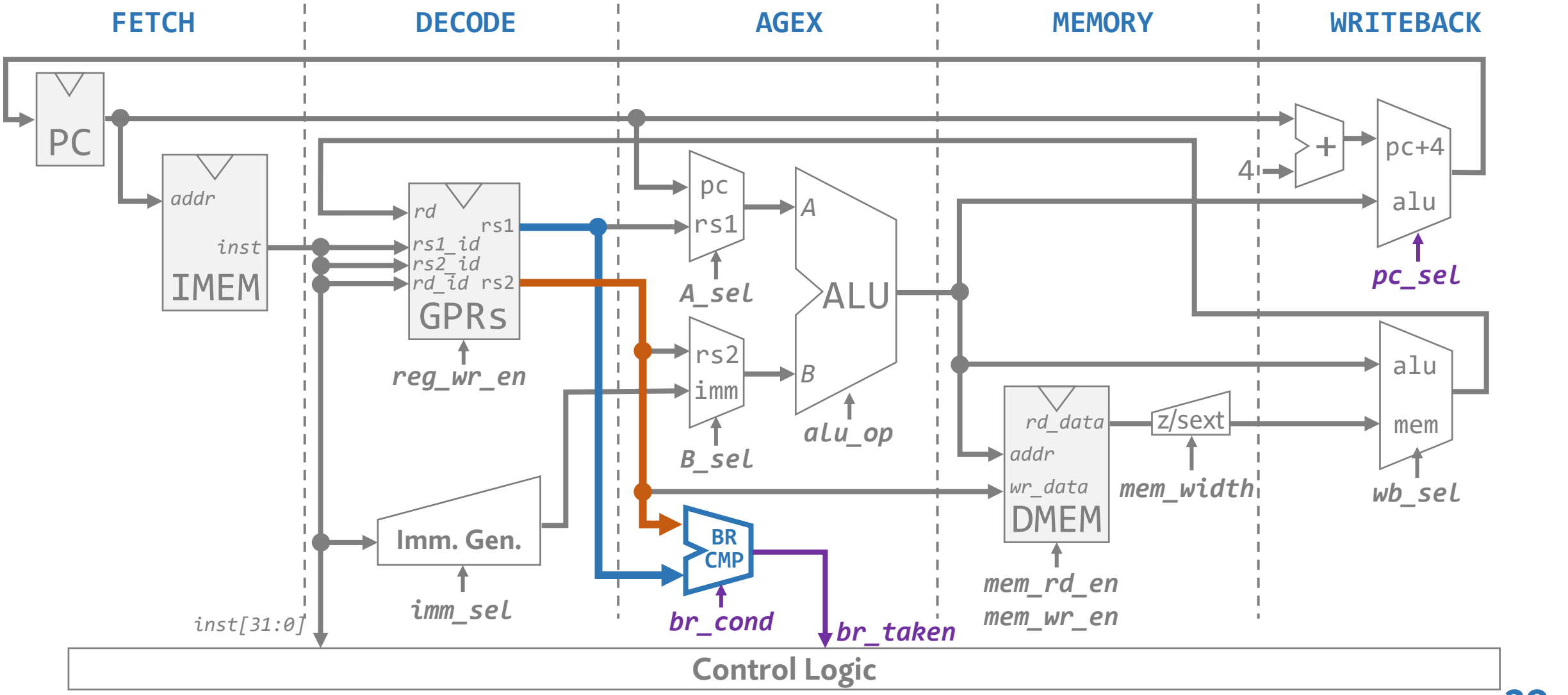
br_cond

```
000 rs1 == rs2
001 rs1 != rs2
100 rs1 < rs2
101 rs1 >= rs2
110 rs1 < rs2 (unsigned)
111 rs1 >= rs2 (unsigned)
```



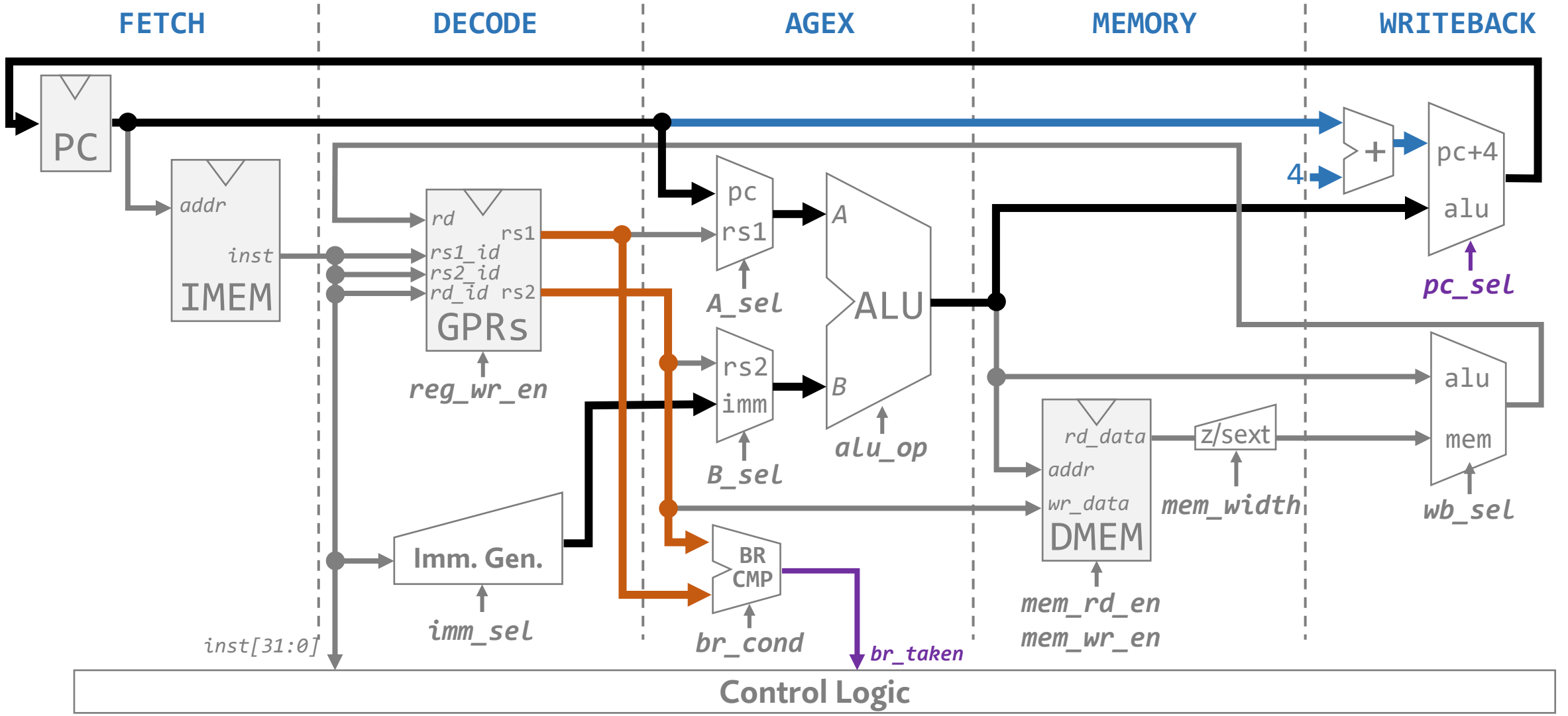
2. Use BR CMP for cond

State Element	Action
PC	PC += taken ? sext(imm[12:0]) : 4
GPRs	READ(R[rs1] & R[rs2])
DMEM	-



Full BRANCH Datapath

State Element	Action
PC	PC += taken ? sext(imm[12:0]) : 4
GPRs	READ(R[rs1] & R[rs2])
DMEM	-



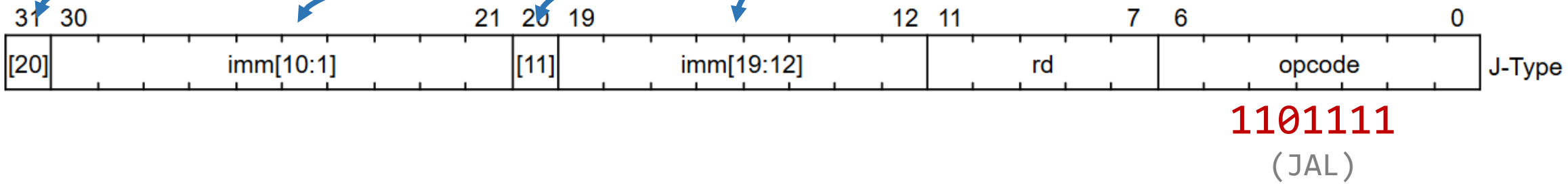
Agenda

- Memory Operations
  - I-Type: Loads
  - S-Type: Stores

- Control Flow Operations
  - B-Type: Branches
  - **Jumps and J-Type**
  - U-Type: AUIPC and LUI

Jump and Link (J-Type)

```
jal rd, <PC offset>
```



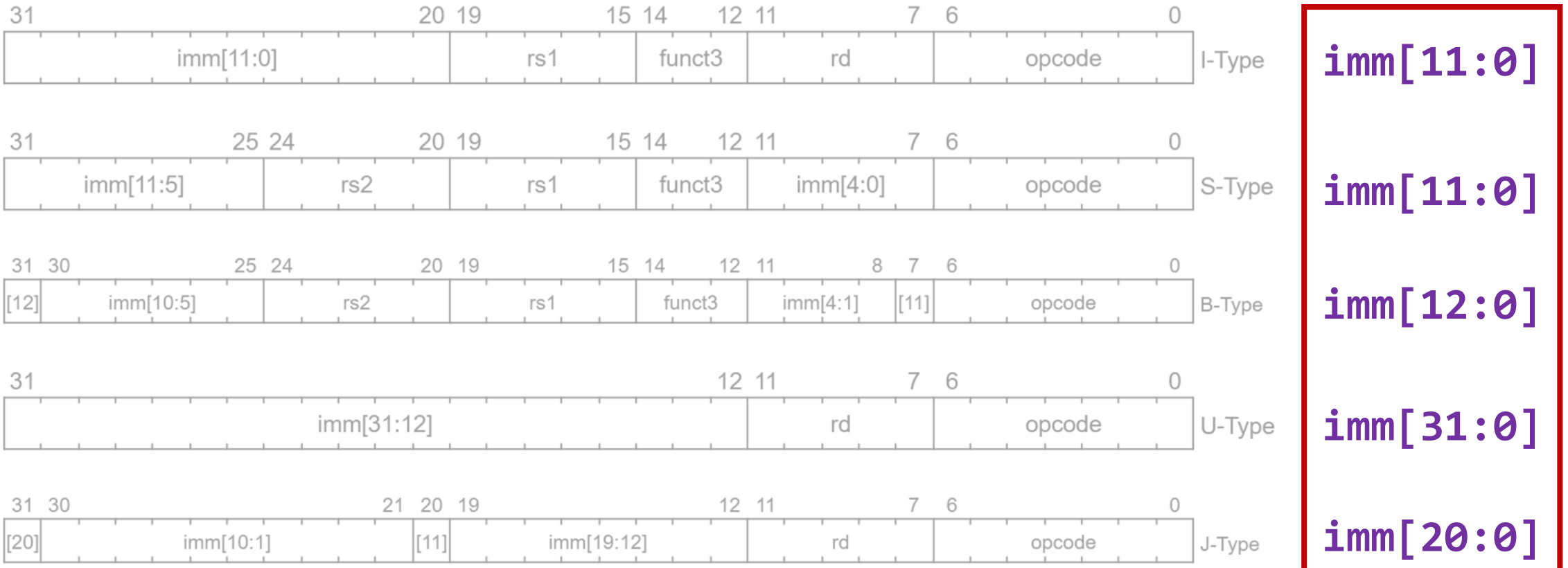
1101111
(JAL)

imm[20:0] = | inst[31] | inst[19:12] | inst[20] | inst[30:21] | 0 |

(new) five-piece immediate!

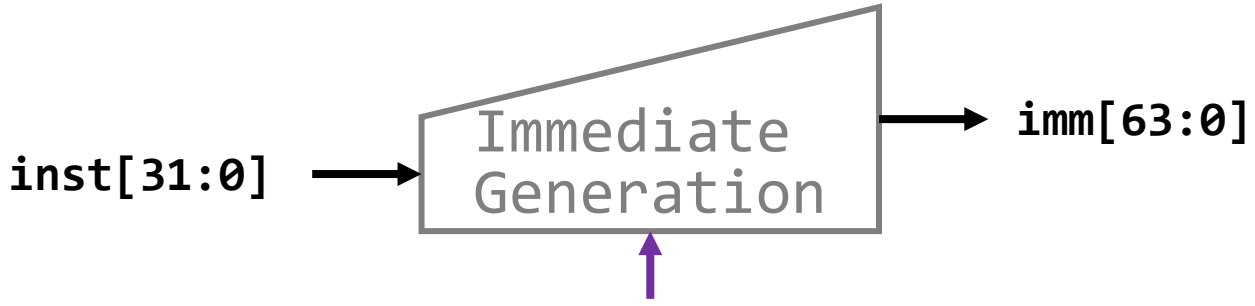
- **{rd, opcode}** are in the same places as R/I/S/B-Type

All These Crazy Immediates!



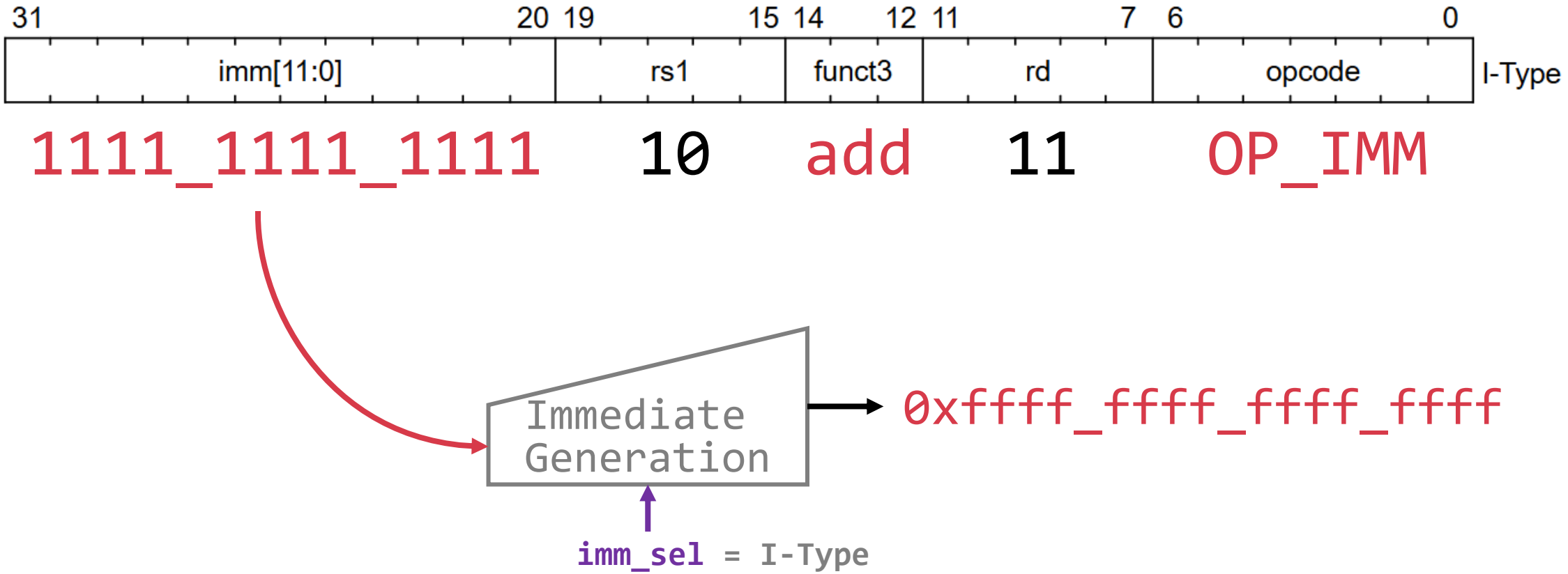

```
imm[11:0]
imm[11:0]
imm[12:0]
imm[31:0]
imm[20:0]
```

All get sign extended to 64 bits



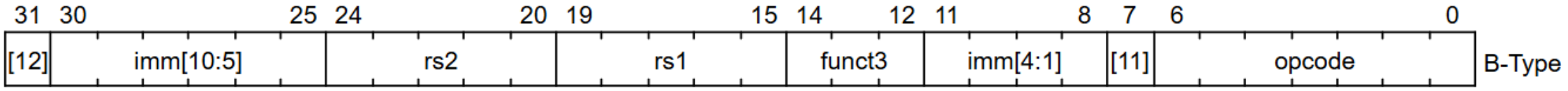
Example: ADDI

```
addi x11, x10, -1
```

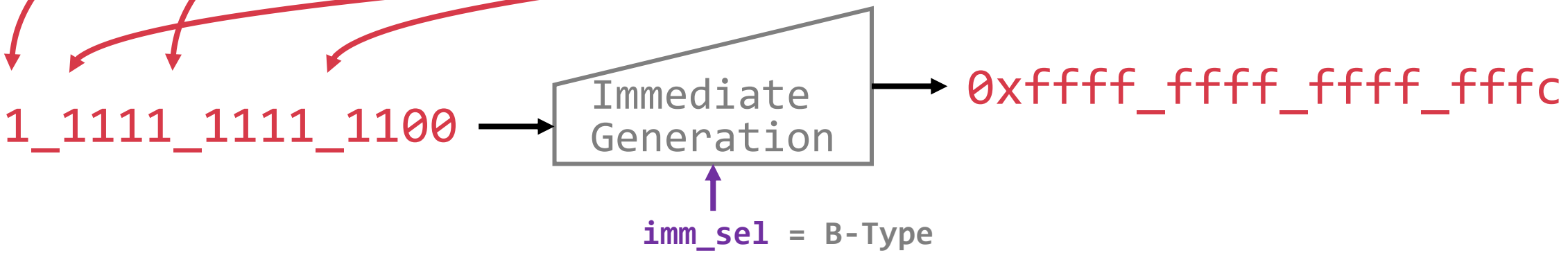


Example: BEQ

```
beq x10, x11, -4
```

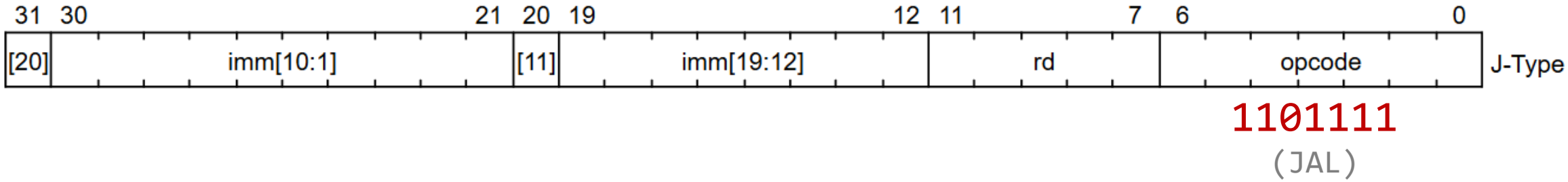


1 111111 11 10 eq 1110 1 BRANCH



Jump and Link (J-Type)

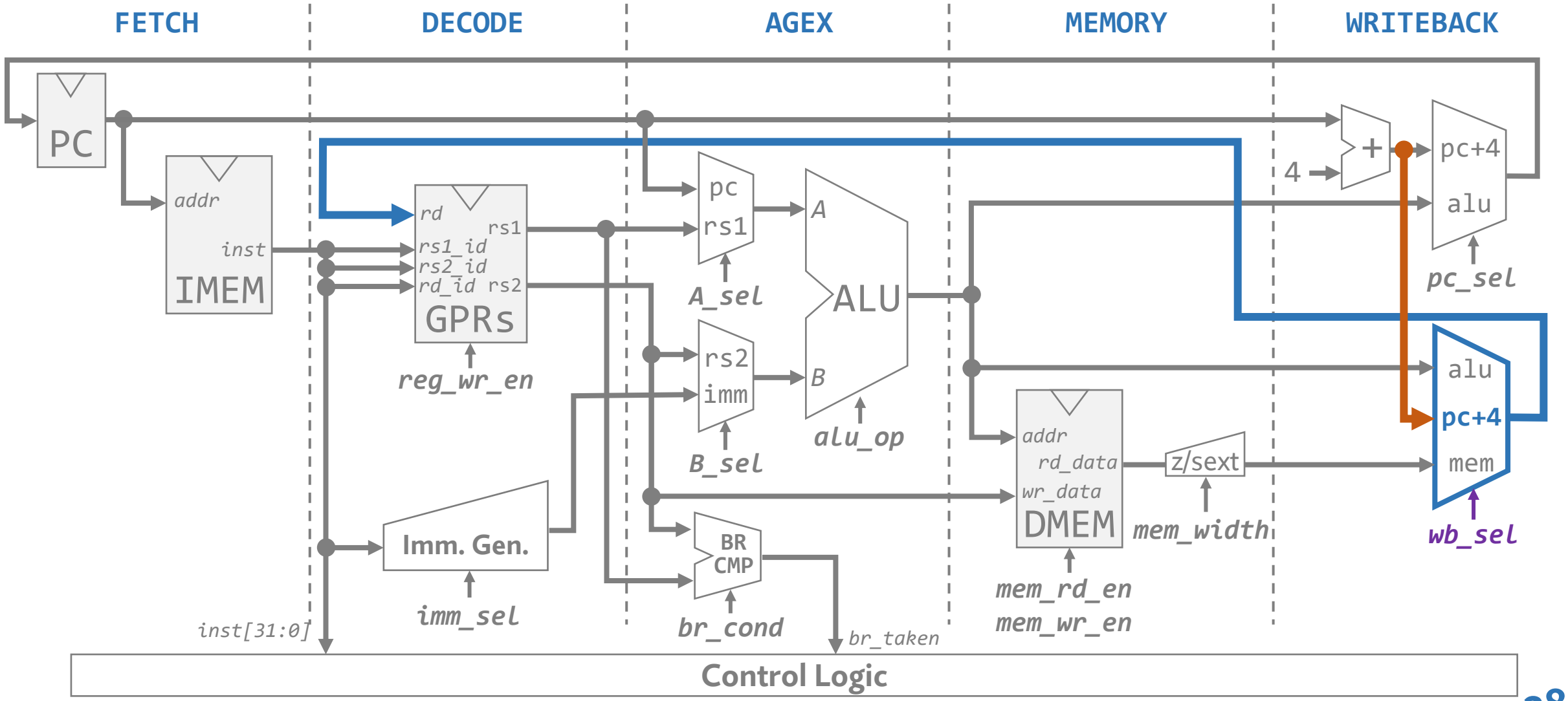
```
jal rd, <PC offset>
```



State Element	Action
PC	PC += sext(imm[20:0])
GPRs	WRITE(R[rd]) *save ra (pc + 4)*
DMEM	-

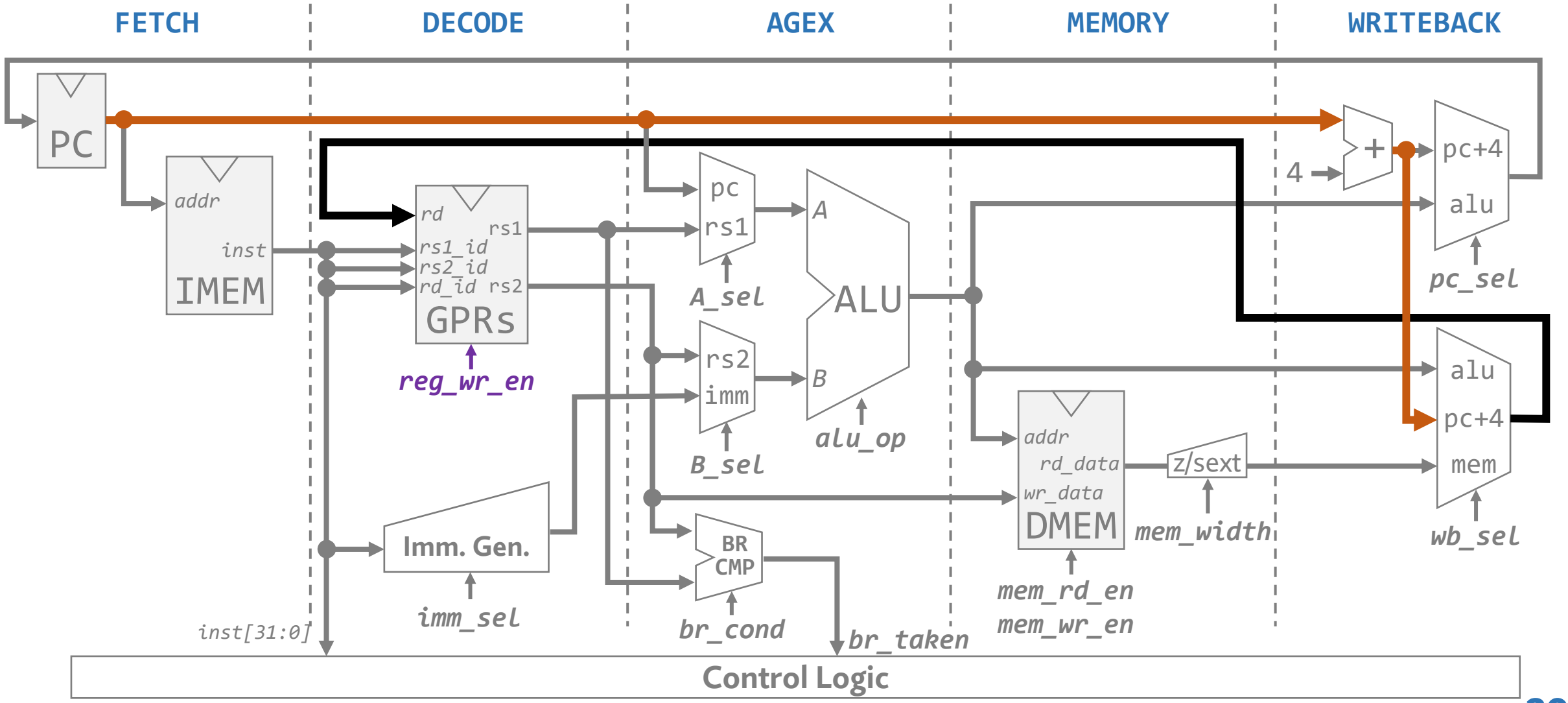
J-Type Datapath Updates

State Element	Action
PC	PC += sext(imm[20:0])
GPRs	WRITE(R[rd])
DMEM	-



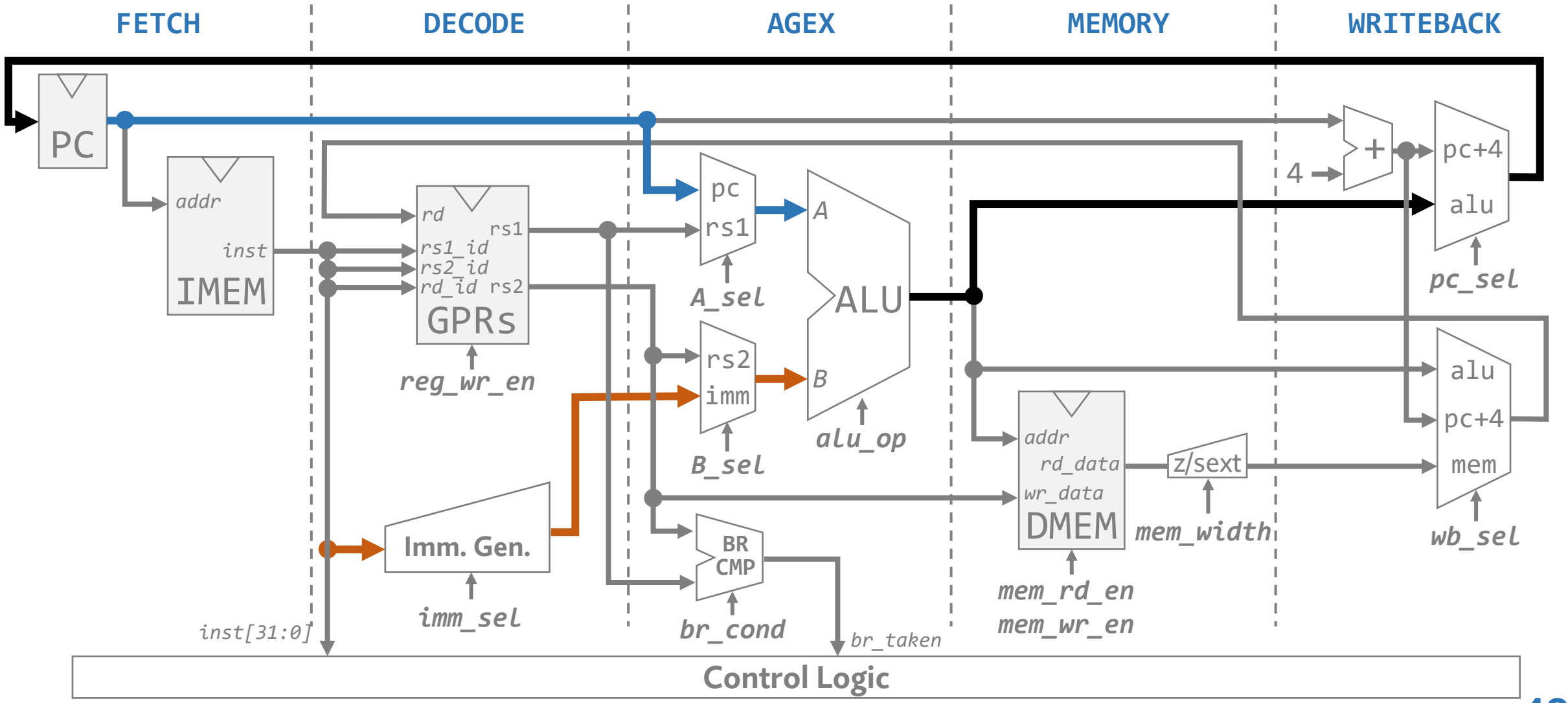
JAL: Saving the Return Address

State Element	Action
PC	PC += sext(imm[20:0])
GPRs	WRITE(R[rd])
DMEM	-



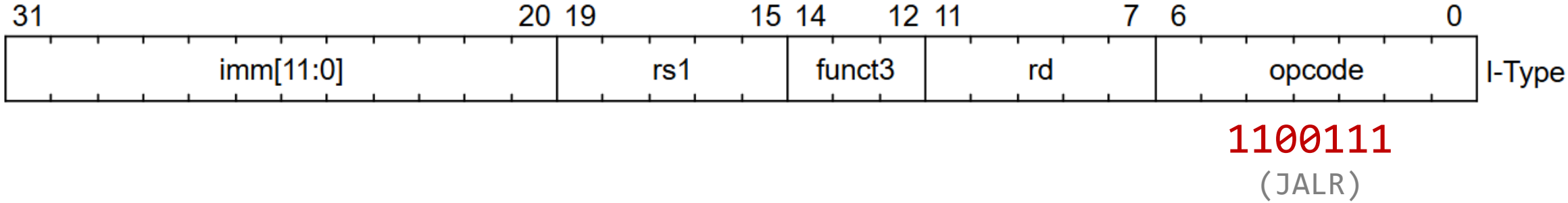
JAL: Updating the PC

State Element	Action
PC	PC += sext(imm[20:0])
GPRs	WRITE(R[rd])
DMEM	-



Jump and Link Register (I-Type)

```
jalr rd, rs1, <PC offset>
```

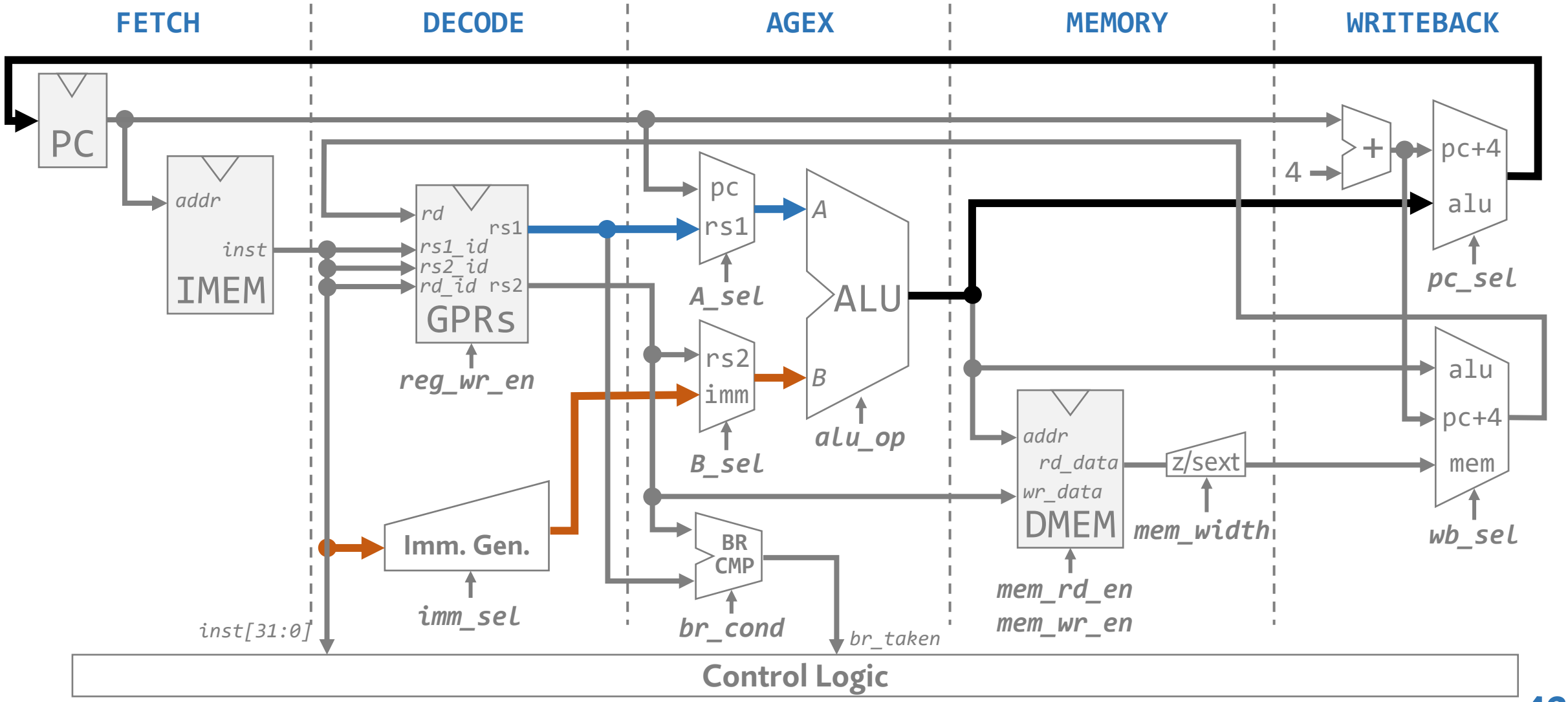


- Our datapath **already handles** I-Type instructions ☺

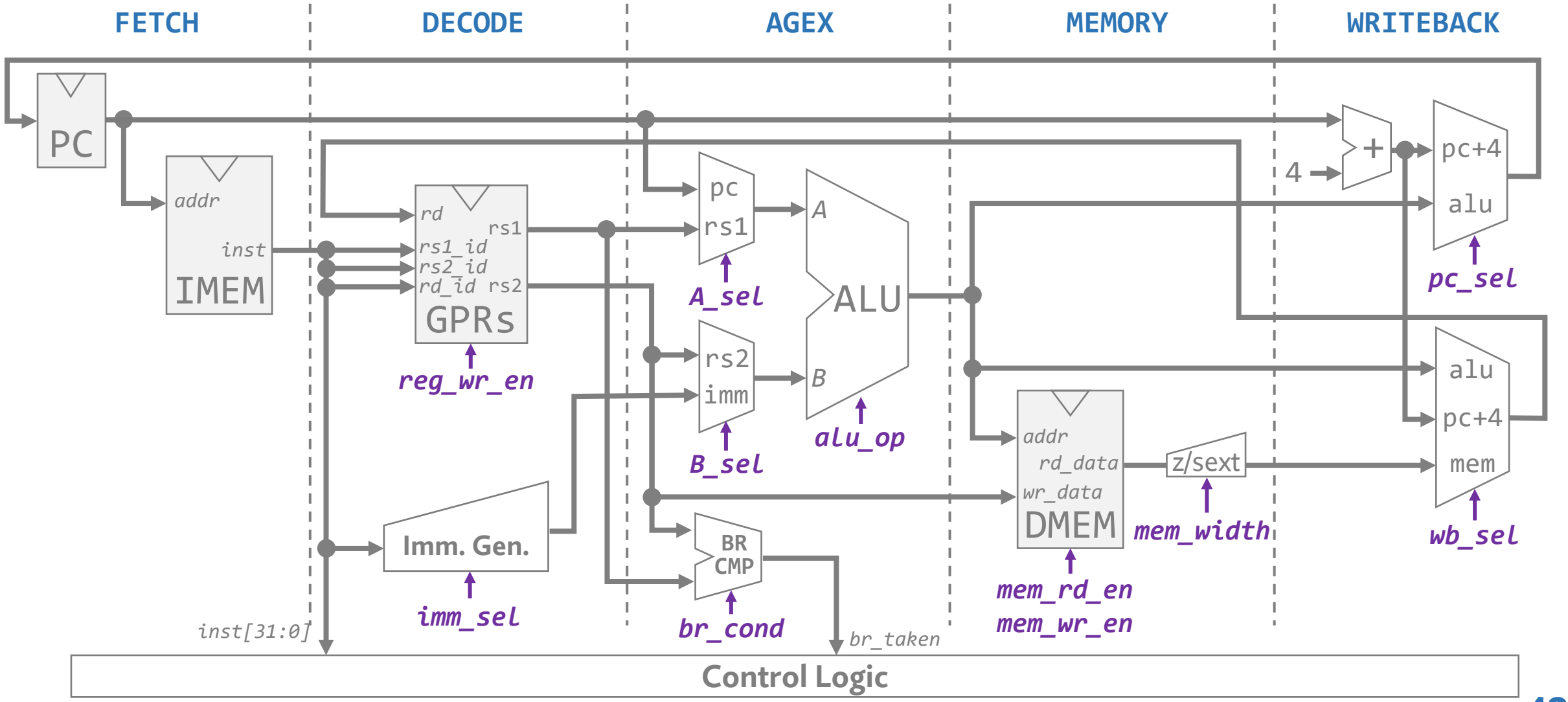
State Element	Action
PC	PC = **R[rs1]** + IMM
GPRs	WRITE(R[rd]) *save ra (pc + 4)*
DMEM	-

JALR: Updating the PC

State Element	Action
PC	PC = R[rs1] + IMM
GPRs	WRITE(R[rd])
DMEM	-



Datapath So Far (Almost Done!)



Agenda

- Memory Operations
  - I-Type: Loads
  - S-Type: Stores
- Control Flow Operations
  - B-Type: Branches
  - Jumps and J-Type
  - **U-Type: AUIPC and LUI**

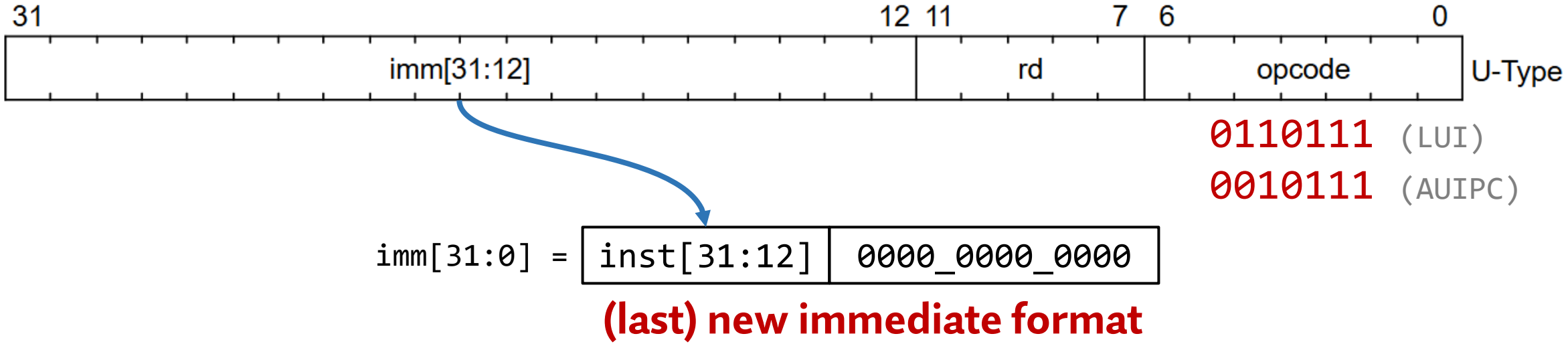
Load Upper Immediate: U-Type

```
lui rd, imm
```

```
rd = sext(imm[31:0])
```

```
auipc rd, imm
```

```
rd = pc + sext(imm[31:0])
```



- **{rd, opcode}** are in the same places as R/I/S/B/J-Type

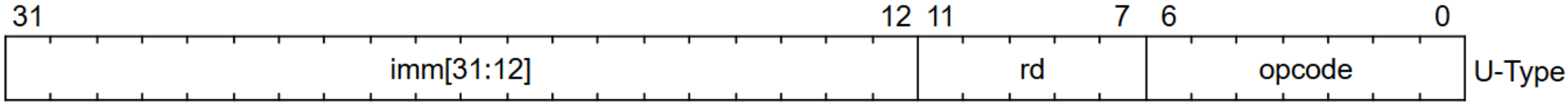
Load Upper Immediate: U-Type

```
lui rd, imm
```

```
rd = sext(imm[31:0])
```

```
auipc rd, imm
```

```
rd = pc + sext(imm[31:0])
```



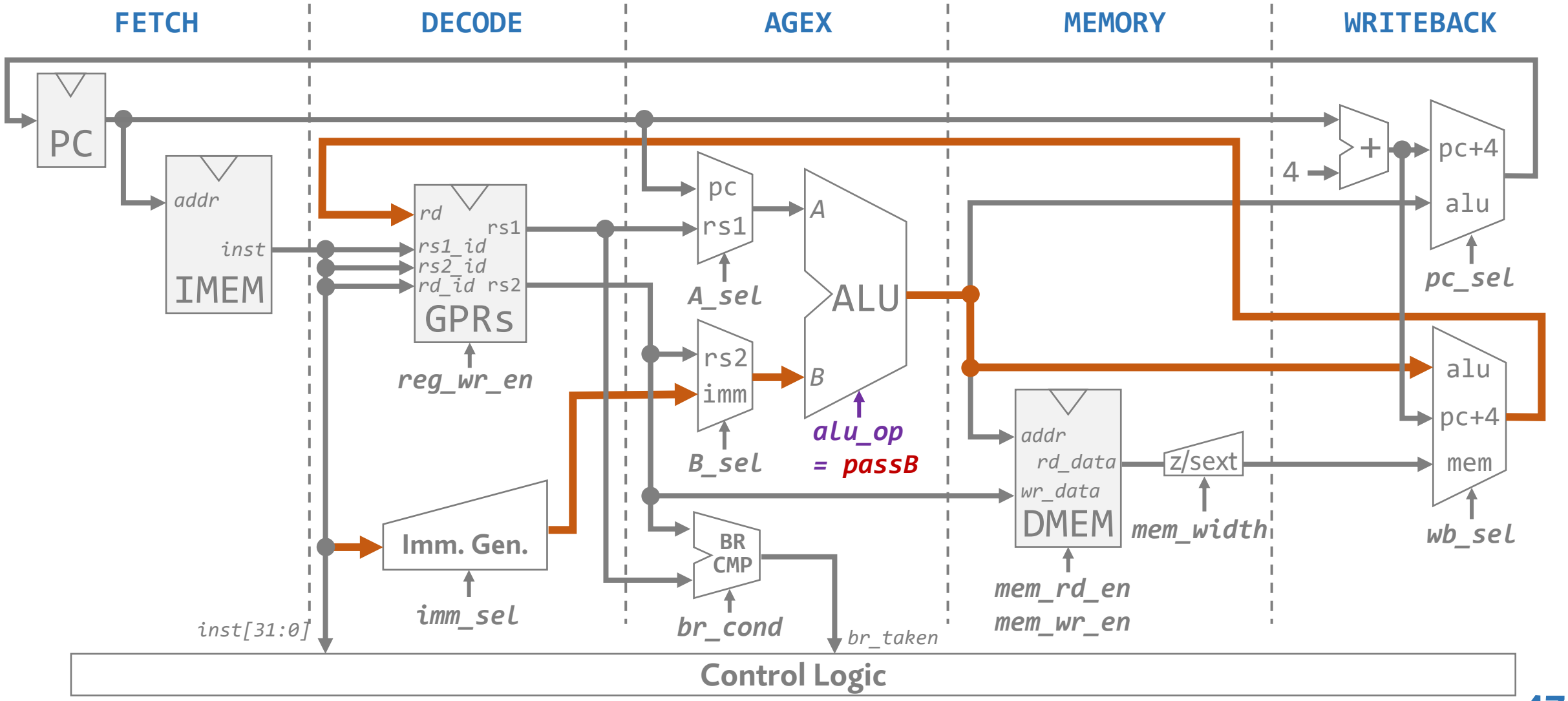
`0110111` (LUI)

`0010111` (AUIPC)

State Element	Action
PC	PC += 4
GPRs	WRITE(R[rd])
DMEM	-

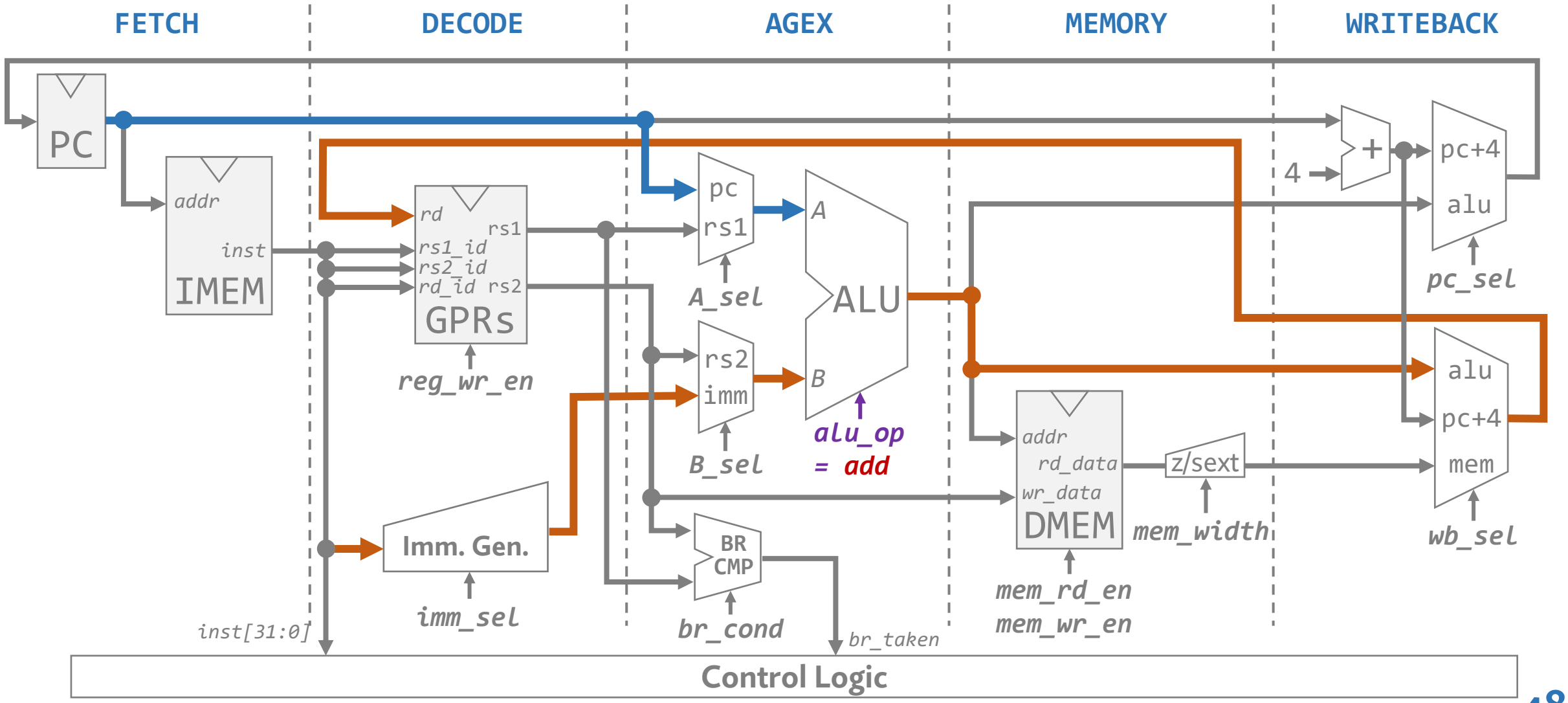
Load Upper Immediate

State Element	Action
PC	PC += 4
GPRs	WRITE(R[rd])
DMEM	-

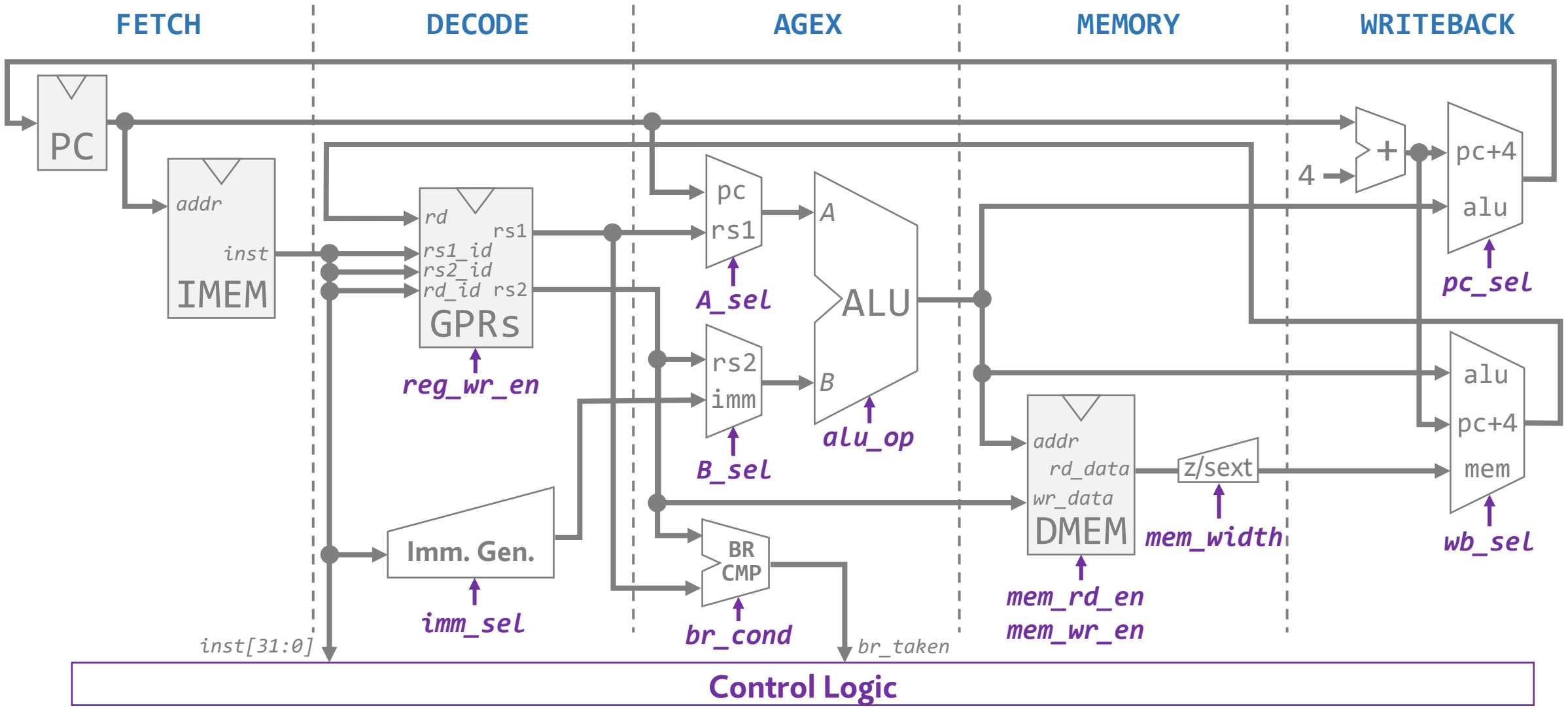


Add Upper Immediate to PC

State Element	Action
PC	PC += 4
GPRs	WRITE(R[rd])
DMEM	-



The Final Datapath



We've Done It: RISC-V CPU!

- We've designed a computer that can execute **any* code you want**
- **Takeaway:** everything boils down to **logic operations on numbers**

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
        print("Hello, World");
        return EXIT_SUCCESS;
}
```

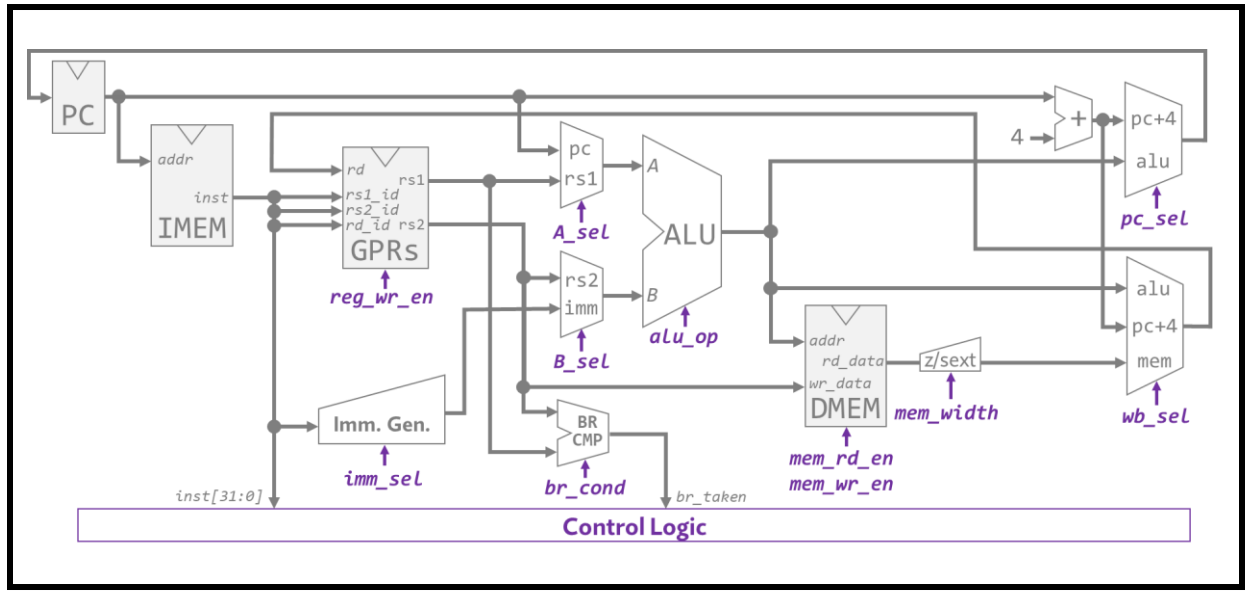
hello.c

```
main:
        addi sp,sp,-32
        sd ra,24(sp)
        sd s0,16(sp)
        addi s0,sp,32
        mv a5,a0
        sd a1,-32(s0)
        sw a5,-20(s0)
        lla a0,.LC0
        call puts
        li a5,0
        mv a0,a5
        ld ra,24(sp)
        ld s0,16(sp)
        addi sp,sp,32
        jr ra
```

hello.S

```
00001050: 4743 433a 2028 5562 756e 7475 2031 312e  GCC: (Ubuntu 11.
00001060: 342e 302d 3175 6275 6e74 7531 7e32 322e  4.0-1ubuntu1~22.
00001070: 3034 2920 3131 2e34 2e30 0041 3200 0000  04) 11.4.0.A2...
00001080: 7269 7363 7600 0128 0000 0005 7276 3634  riscv..(....rv64
00001090: 6932 7030 5f6d 3270 305f 6132 7030 5f66  i2p0_m2p0_a2p0_f
000010a0: 3270 305f 6432 7030 5f63 3270 3000 002e  2p0_d2p0_c2p0...
000010b0: 7368 7374 7274 6162 002e 696e 7465 7270  shstrtab..interp
000010c0: 002e 6e6f 7465 2e67 6e75 2e62 7569 6c64  ..note.gnu.build
000010d0: 2d69 6400 2e6e 6f74 652e 4142 492d 7461  -id..note.ABI-ta
000010e0: 6700 2e67 6e75 2e68 6173 6800 2e64 796e  g..gnu.hash..dyn
000010f0: 7379 6d00 2e64 796e 7374 7200 2e67 6e75  sym..dynstr..gnu
00001100: 2e76 6572 7369 6f6e 002e 676e 752e 7665  .version..gnu.ve
00001110: 7273 696f 6e5f 7200 2e72 656c 612e 6479  rsion_r..rela.dy
00001120: 6e00 2e72 656c 612e 706c 7400 2e74 6578  n..rela.plt..tex
00001130: 7400 2e72 6f64 6174 6100 2e65 685f 6672  t..rodata..eh_fr
00001140: 616d 655f 6864 7200 2e65 685f 6672 616d  ame_hdr..eh_fram
00001150: 6500 2e70 7265 696e 6974 5f61 7272 6179  e..preinit_array
00001160: 002e 696e 6974 5f61 7272 6179 002e 6669  ..init_array..fi
00001170: 6e69 5f61 7272 6179 002e 6479 6e61 6d69  ni_array..dynami
00001180: 6300 2e64 6174 6100 2e67 6f74 002e 6273  c..data..got..bs
00001190: 7300 2e63 6f6d 6d65 6e74 002e 7269 7363  s..comment..risc
000011a0: 762e 6174 7472 6962 7574 6573 0000 0000  v.attributes....
```

hello



Compare: Another RISC-V Datapath

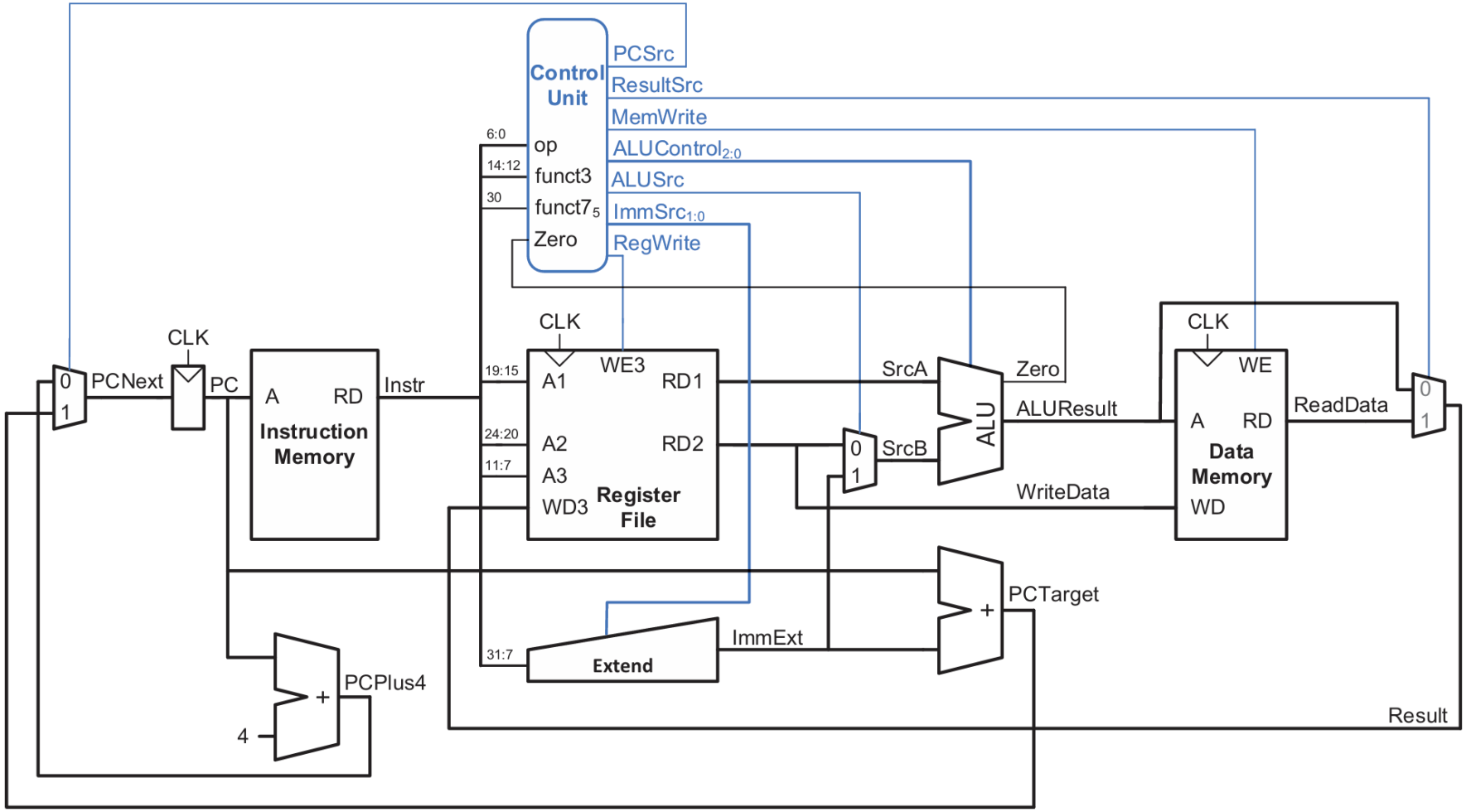


Figure 7.12 Complete single-cycle processor

CS 211: Intro to Computer Architecture

12.2: Single-Cycle RV64I CPU – Mem, Branch, & Jump

Minesh Patel

Spring 2025 – Thursday 17 April