Version 1.08

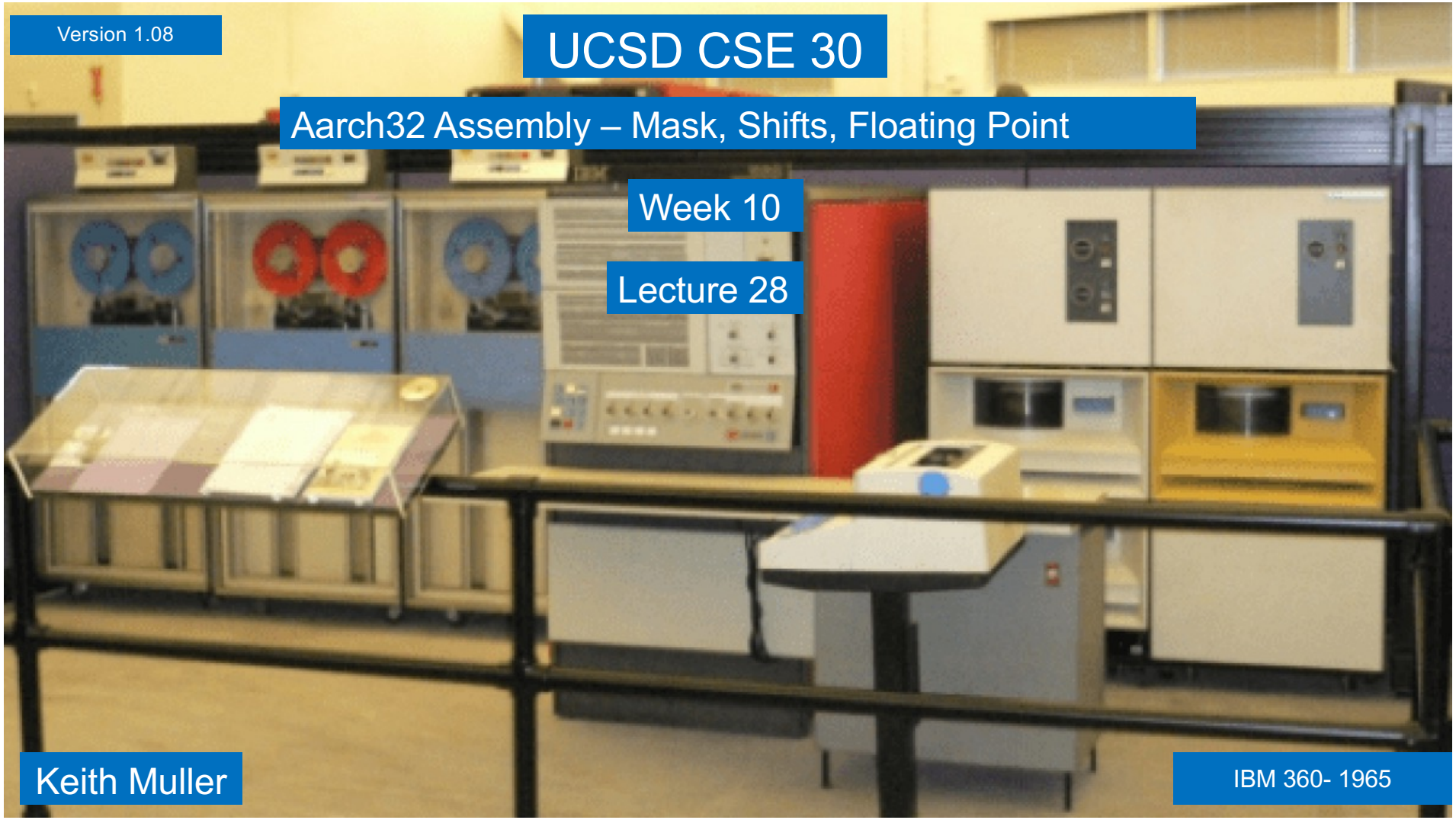
UCSD CSE 30

Aarch32 Assembly – Mask, Shifts, Floating Point

Week 10

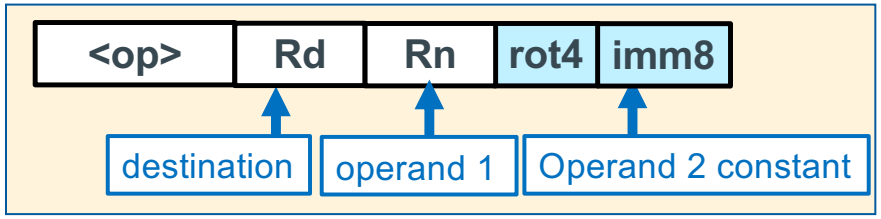
Lecture 28

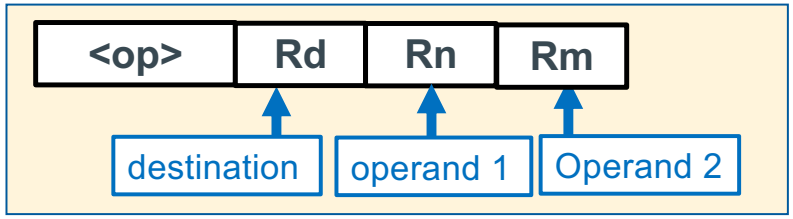
Keith Muller



IBM 360- 1965

Bitwise Instructions






```
<op>  Rd,  Rn, constant    // Rd = Rn <op> constant
<op>  Rd,  constant        // Rd = Rd <op> constant
<op>  Rd,  Rn,  Rm         // Rd = Rn <op> Rm
```

Bytes: 0 <= imm8 <= 255 + values from "rotating" rot 4 bits

Bitwise <op> description	*<op> Syntax*	*Operation*
Bitwise **AND**	`and` R_d, R_n, Op2	$R_d \leftarrow R_n$ & Op2
Bit Clear each bit in Op2 that is a 1, the same bit in R_d, is cleared	`bic` R_d, R_n, Op2	$R_d \leftarrow R_n$ & ~Op2
Bitwise **OR**	`orr` R_d, R_n, Op2	$R_d \leftarrow R_n$ | Op2
Exclusive **OR**	`eor` R_d, R_n, Op2	$R_d \leftarrow R_n$ ^ Op2

x

Masking Summary

Select a field: Use `and` with a mask of one's surrounded by zero's to select the bits that have a 1 in the mask, all other bits will be set to zero

selects this field when used with and

0 0 0 0 0 0 0 0	1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0

selection mask

Clear a field: Use `and` with a mask of zero's surrounded by one's to select the bits that have a 1 in the mask, all other bits will be set to zero

clears this field when used with and

1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0 0	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

clear a field mask

Isolate a field: Use `lsr, lsl, rot` to get a field surrounded by zeros

0 0 0 0 0 0 0 0	*source*	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

lsl to get this edge into msb

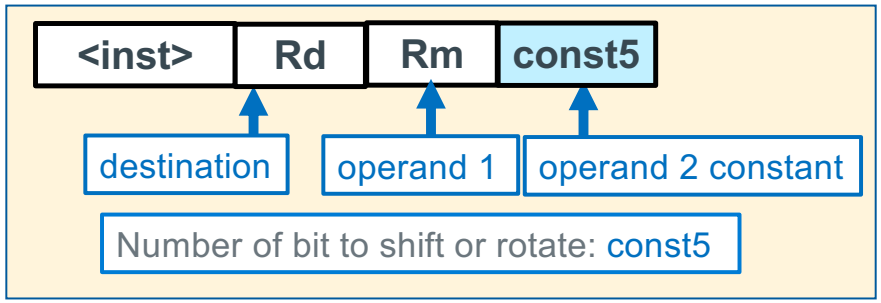
lsr to get this edge into lsb

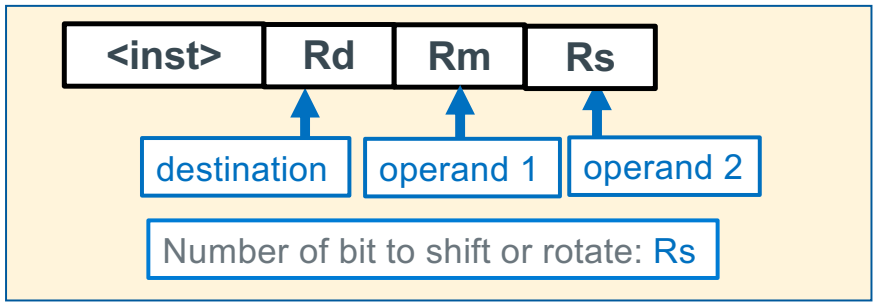
Insert a field: Use `orr` with fields surrounded by zeros

0 0 0 0 0 0 0 0	*source*	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Keep these bits	0 0 0 0 0 0 0 0	Keep these bits

x

Shift and Rotate Instructions



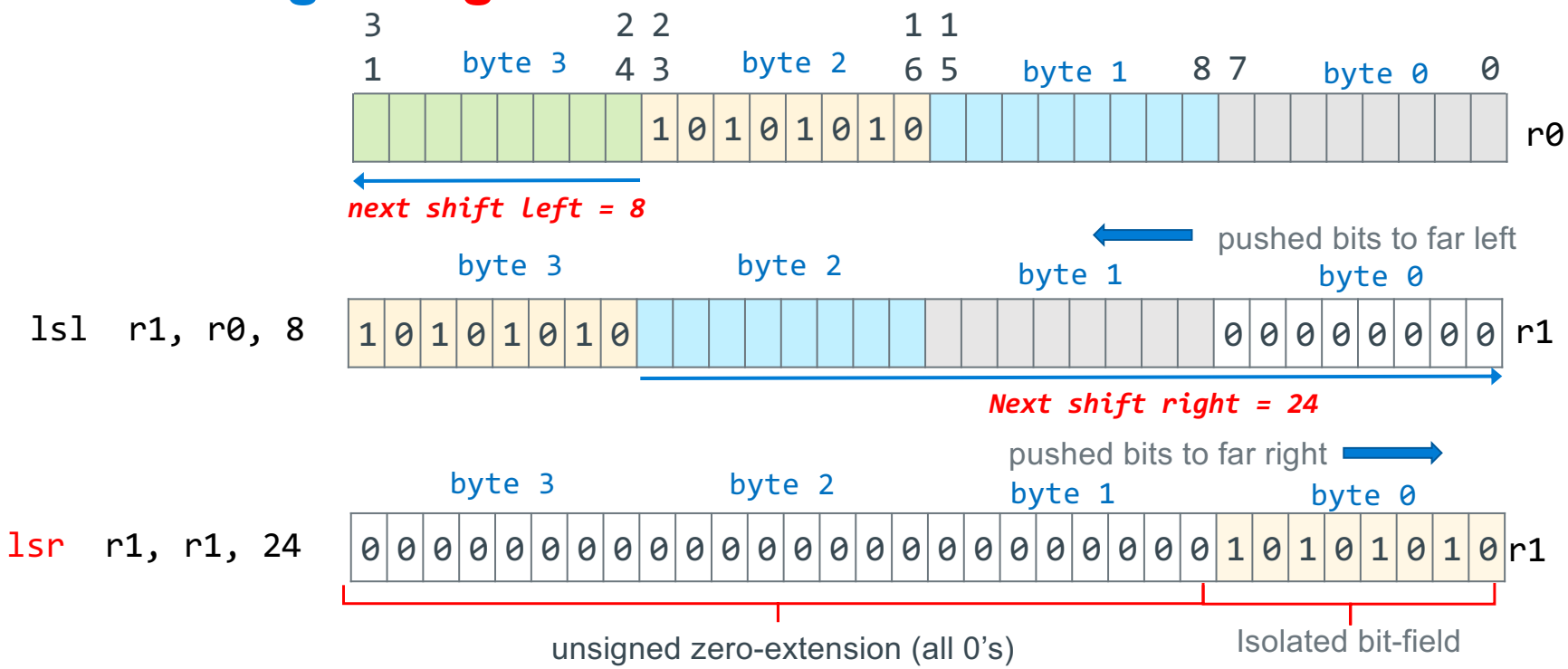


Instruction	Syntax	Operation	Notes	Diagram
Logical Shift Left	LSL R_d, R_m, *const5* LSL R_d, R_m, R_s	$R_d \leftarrow R_m << const5$ $R_d \leftarrow R_m << R_s$	Zero fills shift: 0 - 31	
Logical Shift Right	LSR R_d, R_m, *const5* LSR R_d, R_m, R_s	$R_d \leftarrow R_m >> const5$ $R_d \leftarrow R_m >> R_s$	Zero fills shift: 1 - 32	
Arithmetic Shift Right	ASR R_d, R_m, *const5* ASR R_d, R_m, R_s	$R_d \leftarrow R_m >> const5$ $R_d \leftarrow R_m >> R_s$	Sign extends shift: 1 - 32	
Rotate Right	ROR R_d, R_m, *const5* ROR R_d, R_m, R_s	$R_d \leftarrow R_m$ ror *const5* $R_d \leftarrow R_m$ ror R_s	right rotate rot: 0 - 31	

x

Isolating Unsigned Bitfields

Hint: Useful for PA5



- You can use ror to move the field to the desired location
- Alternative: If you can create an immediate value mask with a data operation like: movn, mov, add, or sub that is often faster

x

C fread/fwrite Example - 1

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#define BFSZ      8192 /* size of read */
int main(void)
{
  char fbuf[BFSZ];
  FILE *fin, *fout;
  size_t readlen;
  size_t bytes_copied = 0;
  retval = EXIT_SUCCESS;

  if (argc != 3){
    fprintf(stderr, "%s requires two args\n", argv[0]);
    return EXIT_FAILURE;
  }
  /* Open the input file for read */
  if ((fin = fopen(argv[1], "r")) == NULL) {
    fprintf(stderr,"fopen for read failed\n");
    return EXIT_FAILURE;
  }
  /*  Open the output file for write */
  if ((fout = fopen(argv[2], "w") == NULL) {
    fprintf(stderr, "fopen for write failed\n");
    fclose(fin);
    return EXIT_FAILURE;
  }
```

To handle bytes moved

```
% ls -ls ZZZ
ls: ZZZ: No such file or directory
% ./a.out cp.c ZZZ
bytes copied: 1122
% ls -ls cp.c ZZZ
8 -rw-r--r--  1 kmuller  staff  1122 Jul  2 08:51 ZZZ
8 -rw-r--r--  1 kmuller  staff  1122 Jul  2 08:49 cp.c
```


x

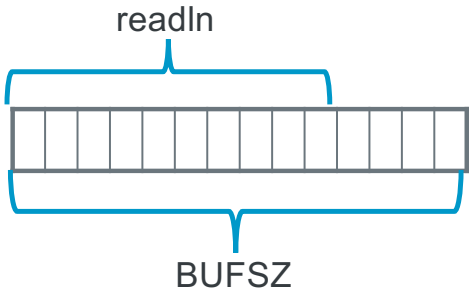
C fread/fwrite Example - 2

```
  /* Read from the file, write to fout */

  while ((readlen = fread(fbuf, 1, BUFSIZ, fin)) > 0) {

    if (fwrite(fbuf, 1, readlen, fout) != readlen) {
       fprintf(stderr, "write failed\n");
        retval =  EXIT_FAILURE;
        break;
    }
    bytes_copied += readlen; //running sum bytes copied
  }

  if (retval == EXIT_FAILURE)
    printf("Failure Copy did not complete only ");
  printf("Bytes copied: %zu\n", bytes_copied);

  fclose(fin);
  fclose(fout);

  return retval;
}
```

By using an element size of 1 with a char buffer, this is byte I/O

Capture the bytes read so you know how many bytes to write

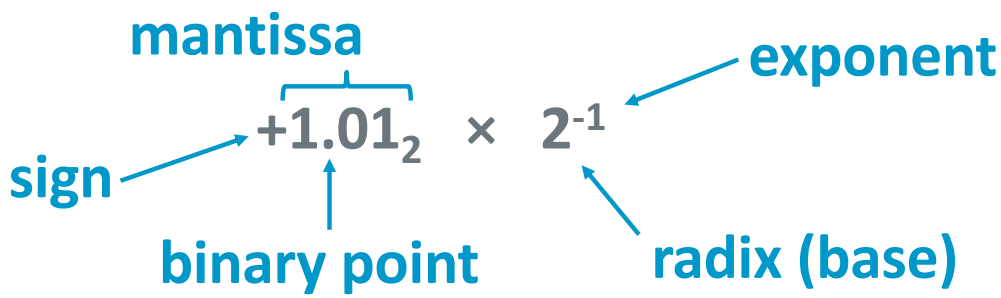
unless file length is an exact multiple of BUFSIZ, the last fread() will always be less than BUFSIZ which is why you write readln



Jargon: the last record is often called the "runt"

x

Scientific Notation Binary



- Computer hardware that supports this is called floating point hardware due to the "floating" of the binary point
- Declare such variable in C as **`float`** (or **`double`**)

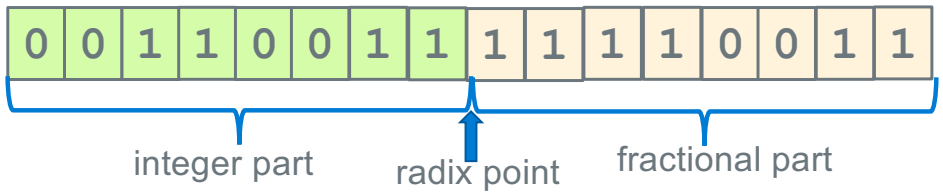
x

Floating Point Representation

- Analogous to scientific notation
- In Decimal:
  - 12000000, is 1.2×10^{7} In C: 1.2e7
  - 0.0000012, is 1.2×10^{-6} In C: 1.2e-6
- In Binary:
  - 11000.000, is 1.1×2^{4}
  - 0.000101, is 1.01×2^{-4}

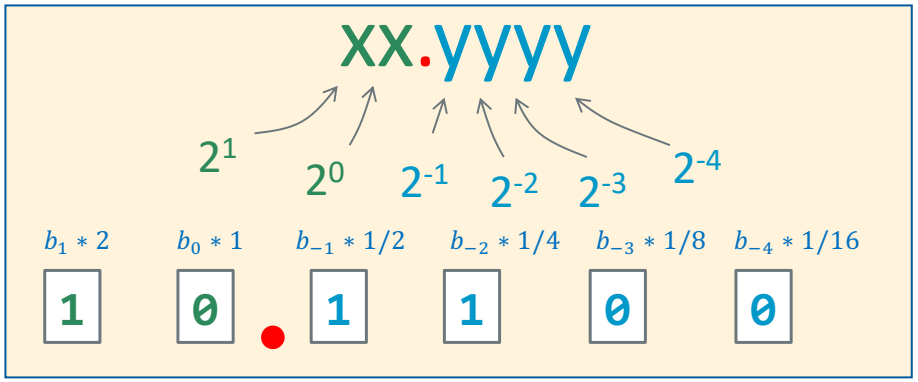
x

Fractional Fixed Point Binary Numbers



Binary	Decimal
2^{-1}	0.5
2^{-2}	0.25
2^{-3}	0.125
2^{-4}	0.0625

- ***"Binary Point",*** like ***decimal point***, signifies boundary between integer and fractional parts
- Bits to right of "binary point" represent fractional powers of 2



Examples:

If the fixed-point format is xxxx.xxxx

$10.1010_2 = 1\times2^1 + 1\times2^{-1} + 1\times2^{-3} = 2.625_{10}$

$100.1_2 = 1\times2^2 + 1\times2^{-1} = 4.5_{10}$

$1.111_2 = 1\times2^0 + 1\times2^{-1} + 1\times2^{-2} + 1\times2^{-3} = 1.875_{10}$

x

Scientific Notation – Base 10

Examples In Base 10:

$42.4 \times 10^5 = 4.24 \times 10^6$

$324.5 \times 10^5 = 3.245 \times 10^7$

$0.624 \times 10^5 = 6.24 \times 10^4$

Observation on base 10:

- We usually adjust the exponent until we get down to one digit to the left of the decimal point
- The mantissa is normalized

x

Normalized Scientific Notation

- Convert from scientific notation to fixed binary point
- Perform the multiplication by shifting the decimal until the exponent disappears

Binary	Decimal
2^{-1}	0.5
2^{-2}	0.25
2^{-3}	0.125
2^{-4}	0.0625

  - Example: $1.011_2 \times 2^4 = 10110_2 = 22_{10}$
  - Example: $1.011_2 \times 2^{-2} = 0.01011_2 = 0.34375_{10}$
- Convert from binary point to *normalized* scientific notation
  - Distribute out exponents until binary point is to the **right of a single digit**
  - Example: $1101.001_2 = 1.101001_2 \times 2^3$

x

Scientific Notation – Base 2

In Base 2:

10.1	x 2^5	= 1.01	x 2^6
1011.1	x 2^5	= 1.0111	x 2^8
0.110	x 2^5	= 1.10	x 2^4

- **Normalizing with base 2 :**
- There is *always* a 1 to the **left of the decimal point**!
- The 1 is **called the hidden bit**
- We do not have to store the hidden bit since it is there in every normalized mantissa
- "Hidden bit" allows number to have **One additional digit for <u>increased</u> precision**

- **Conversion Rules**
- Adjust x to always be in the format **1.XXXXXXXXX… (mantissa is normalized)**
- Mantissa portion ONLY **encodes** what is *to the **right*** of the decimal point
- **Mantissa encoding is 1.[FRACTION BINARY DIGITS]**
- **But only [FRACTION BINARY DIGITS] are stored**

x

Floating Point Numbers: Implementation Approach

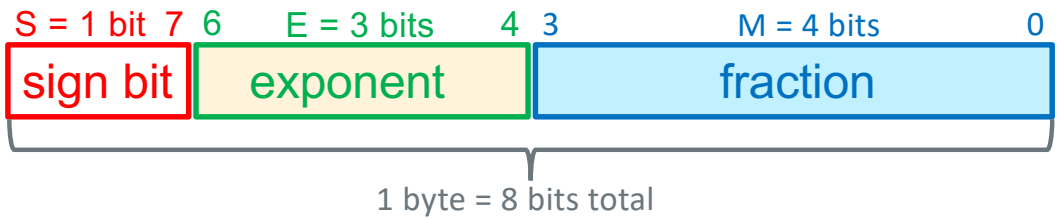
- Supports a wide range of numbers, but limited precision
- Represent scientific notation numbers like 1.202×10^6

$$(-1)^S \; M \; 2^E$$

- **Sign bit** (a single bit): 0 positive, 1 negative
- **Exponent**: **encoding** of E above (it is NOT E directly represented in binary)
- **Fraction**: **encoding** of M above (it is NOT M directly represented in binary)

x

Floating Point Number in a Byte (Not A Real Format)



- **Mantissa encoding: = 1.[xxxx] encoded as an unsigned value**
- **Exponent encoding:** 3 bits encoded as an unsigned value using bias encoding
  - **Use the following variation of Bias encoding = $(2^{E-1} - 1)$**
  - 3 bits for the bias we have $2^{3-1} - 1 = 2^2 - 1$ = a bias of 3
  - **With a Bias of 3**: positive and negative numbers range: small to large is: 2^{-3} to 2^4

Actual	-3	-2	-1	0	1	2	3	4
Bias	+ 3	+ 3	+ 3	+ 3	+ 3	+ 3	+ 3	+3
Biased	0	1	2	3	4	5	6	7

x

Floating Point Number (8-bits) Number Range: 2^{-3} to 2^{4}

S = 1 bit	E = 3 bits	M = 4 bits
sign bit	exponent	fraction

S = 1 bit	E = 3 bits	M = 4 bits
0	000	0000

0.0 <u>Special case</u> in this simple model we <u>do not</u> put back the "hidden bit"

S = 1 bit	E = 3 bits	M = 4 bits
0	000	0001

Smallest Non-zero Positive
0.0010001 = 1/8 + 1/128 = 0.1328125 base 10

S = 1 bit	E = 3 bits	M = 4 bits
0/1	111	1111

Largest Positive/Negative
1.1111 x 2^{4} = 11111 = 31 base 10

S = 1 bit	E = 3 bits	M = 4 bits
1	000	0000

Smallest (closest to zero) Number
1.0000 x 2^{-3} = 0.001000 = 1/8 = -0.125 base 10

Note: Orange is hidden bit added back

x

Decimal to Float

7	6 — Bias of 3 — 4	3 — 0
s	exponent (3 bits)	fraction (4 bits)

Binary	Decimal
2^{-2}	0.25
2^{-3}	0.125

Step 1: convert from base 10 to binary (absolute value)

`-0.375(decimal) = 0000.0110`$_{\text{base 2}}$

Step 2: Find out how many places to shift to get the number into the normalized 1.xxxx mantissa format

$0000.0110_2 = 1.1000 \times (2^{-2})_{\text{base 10}}$

exponent: $-2_{10} + \text{bias of } 3_{10} = 1_{10} = 0b001$ for the exponent (after adding the bias)

Step 3: Use as many digits as possible to the right of the decimal point in the fractional .xxxx part

`1.1000`

Step 4: Sign bit

positive sign bit is 0

negative sign bit is 1

s	exponent	fraction
1	0b001	0b1000
0x9		0x8

= 0x98

Float to Decimal

7	6 … Bias of 3 … 4	3 … 0
s	exponent (3 bits)	fraction (4 bits)

Step 1: Break into binary fields

`0x45 =`

0x4		0x5
s	exponent	fraction
0	0b100	0b0101

Step 2: Extract the unbiased exponent

$0b100 = 4_{base\ 10} - \text{bias of } 3_{10} = 1_{10}$ for the exponent (bias removed)

Step 3: Express the mantissa (restore the hidden bit)

1.0101

Step 4: Apply the **unbiased** exponent

$1.0101_{base\ 2} \times (2^1)_{base\ 10} = 10.101$

Step 5: Convert to decimal

$10.101 = 2.625_{base\ 10}$

Step 6: Apply the Sign

$+\ 2.625_{base\ 10}$

Binary	Decimal
2^{-1}	0.5
2^{-2}	0.25
2^{-3}	0.125
2^{-4}	0.0625

x

IEEE “754” Floating Point Double and Single Precision

Single Precision (C float)

31	30 … 23	22 … 0
sign	Exponent (8 bits) Uses a Bias of 127	fraction (23 bits)

$$Bias\ is\ (2^{8-1} -)\ = 127$$

$$single\ precision\ floating\ point\ number = (-1)^{s}\ x\ 2^{E-127} \text{ x } 1.\text{fraction}$$

Double Precision (C Double)

63	62 … 52	51 … 0
sign	Exponent (11 bits) Uses a Bias of 1023	fraction (52 bits)

$$bias\ is\ (2^{11-1}\ - 1)\ = 1023$$

$$double\ precision\ floating\ point\ number = (-1)^{s}\ x\ 2^{E-1023} \text{ x } 1.\text{fraction}$$

x

Decimal to IEEE Single Precision Float

31	30 … 23	22 … 0
sign	Exponent (8 bits) Bias is 127	fraction (23 bits)

Binary	Decimal
2^{-2}	0.25
2^{-3}	0.125

Step 1: convert from base 10 to binary (absolute value)

`-13.375(decimal) = 1101.0110`

Step 2: Find out how many places to shift to get the number into the normalized 1.xxxx mantissa format

$1101.0110 = 1.1010110 \times (2^3)_{\text{base 10}}$

`3 + bias of 127 = 130 for the exponent = 0b1000 0010`

Step 3: Use as many digits that fit to the right of the decimal point in the fractional .xxxx part (0 pad)

`1.1010110 0000 0000 0000 0000`

Step 4: If the sign is positive sign bit is 0, otherwise it is 1

s	exponent			fraction					
1	100	0001	0	101	0110	0000	0000	0000	0000
	0xc	0x1	0x5	0x6	0x0	0x0	0x0	0x0	= 0xc1560000

x

IEEE Single Precision Float to Decimal

31	30 … 23	22 … 0
sign	Exponent (8 bits) Bias is 127_{10}	fraction (23 bits)

Step 1: Break into binary fields and expand as needed

`0xc0b00000 =`

0xc	0x0	0xb	0x0	0x0	0x0	0x0	0x0
1 100	0000	1 011	0000	0000	0000	0000	0000

Step 2: Find the exponent

$0b10000001 = 129_{base\ 10}$ − bias of 127_{10} = 2_{10} exponent with bias added

Step 3: Express the mantissa (restore the hidden bit)

`1.0110`

Step 4: Apply the exponent

$1.0110 \times (2^2)_{base\ 10} = 101.10$

Step 5: Convert to decimal

`101.10 = 5.5`

Step 6: Apply the Sign

`-5.5`

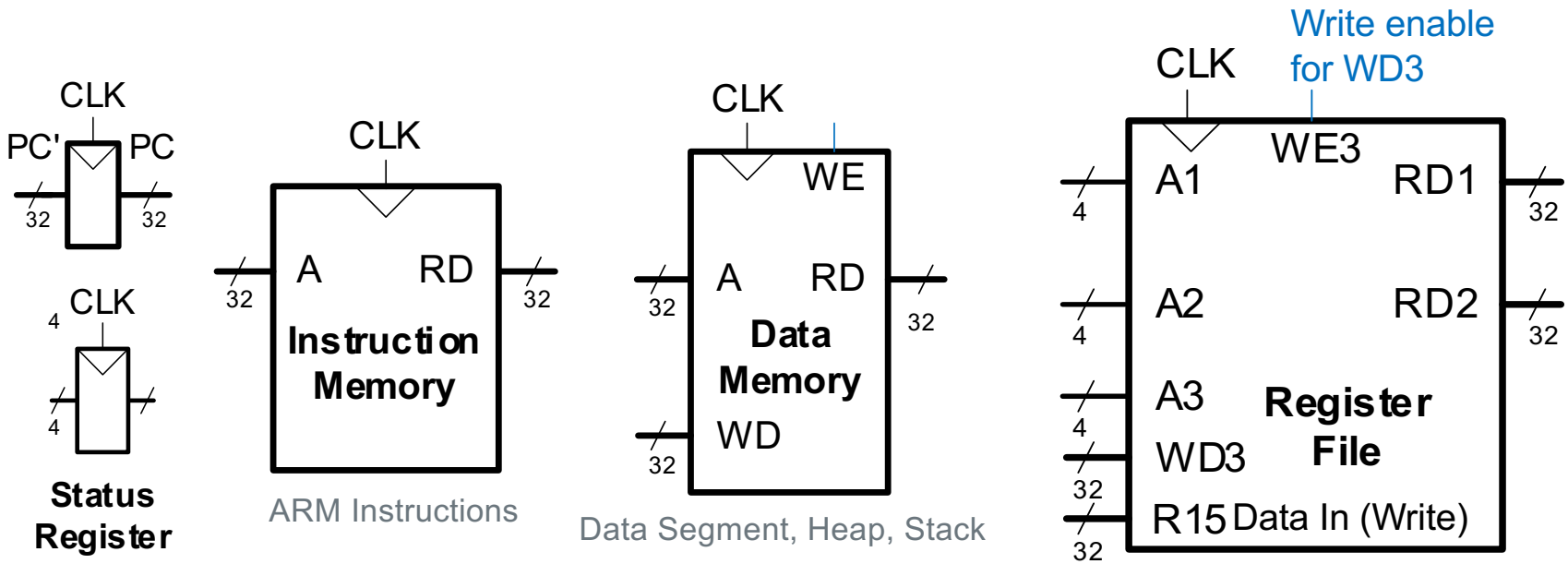
Binary	Decimal
2^{-1}	0.5
2^{-2}	0.25
2^{-3}	0.125
2^{-4}	0.0625

x

Hardware – Not On Final – FYI Only

Primary Processor State Elements

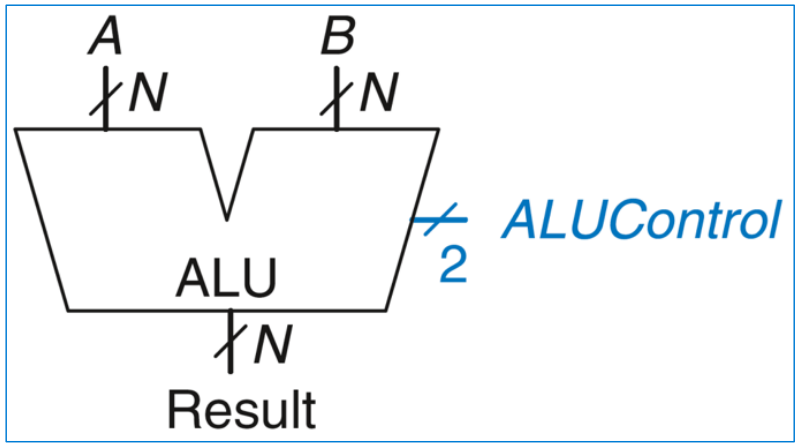
- **These major elements contain the current *state* of a processor:**
- For purpose of class, instructions and data are shown stored using different memories – cse141/cse120 subjects



x

Simple ALU (Arithmetic Logic Unit)

- Processors contain a special logic block called the Arithmetic Logic Unit or ALU
- The ALUop control lines (two in this simplistic case) specify the actual operation to be performed
- For our ARM CPU: N is 32



ALU Control Lines ALUcontrol (1:0)	Function Performed
00	ADD Result, A, B
01	SUB Result, A, B
10	AND Result, A, B
11	OR Result, A, B

Example: Perform *A* OR *B*

ALUControl = 11

$Result = A \text{ OR } B$

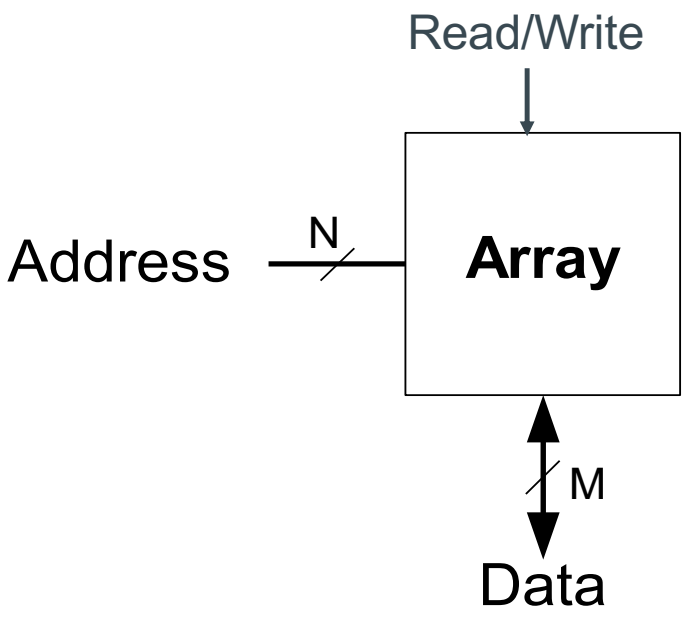
x

Memory Arrays

- Efficiently store large amounts of data
- Different types based on the technology used in the storage element. Examples:
  - Dynamic random-access memory (DRAM)
  - Static random-access memory (SRAM)
  - NVM (non-volatile memory, flash, Intel Optane, etc.)

Interfaces: Address & Data, R/W

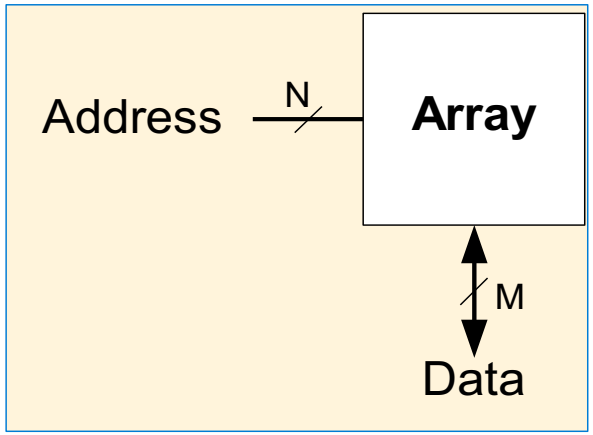
- *M*-bit data value read/written at each unique *N*-bit address

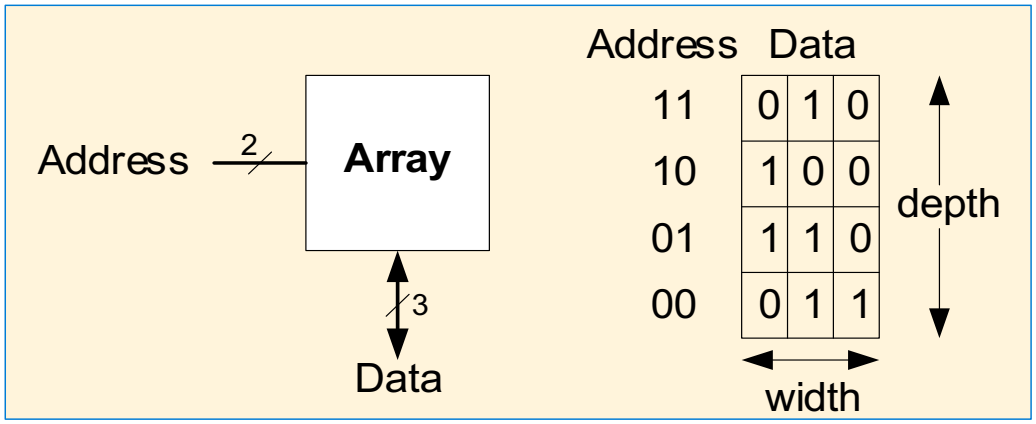


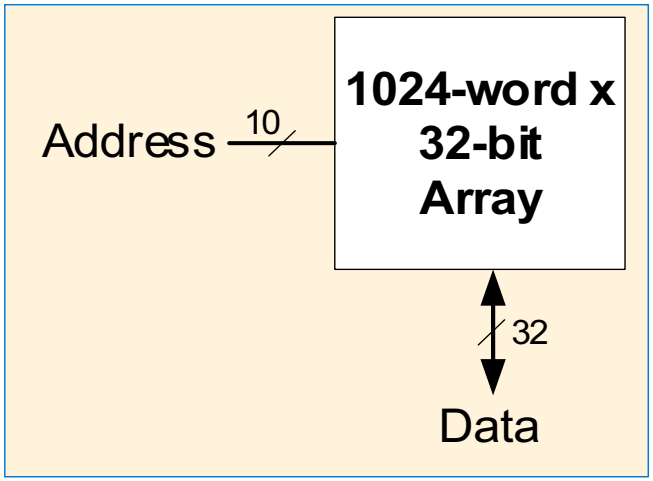
x

Memory Arrays

- 2-dimensional array of bit cells
  - Each bit cell stores one bit of data
- *N* address bits and *M* data bits:
  - 2^N rows and *M* columns
  - **Depth:** number of rows (number of words)
  - **Width:** number of columns (size of word)
  - **Array size:** depth × width = $2^N \times M$



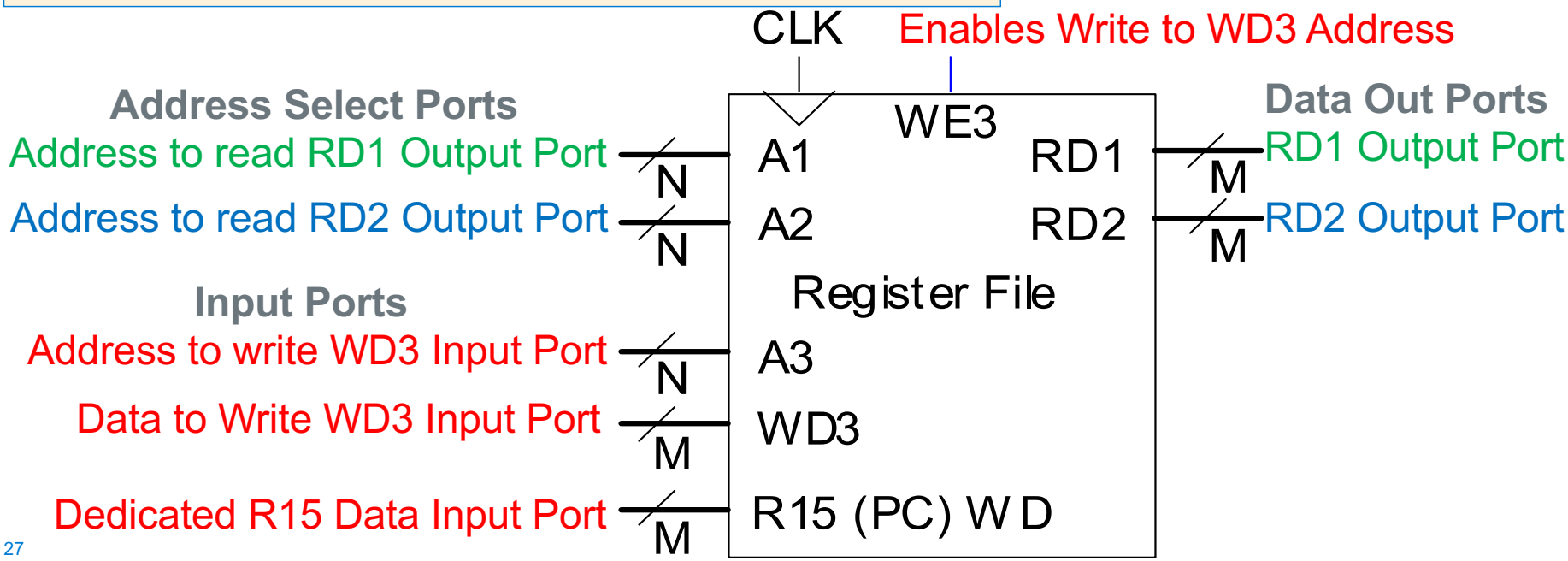




x

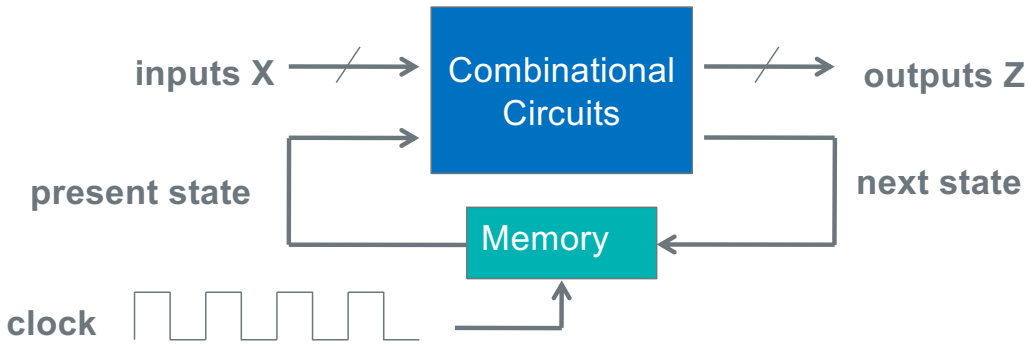
Multi-Ported Register File (Memory)

- **Port:** address/data pair
- 4-ported memory (N address bits, M data bits)
  - 2 read data (out) ports (A1/RD1, A2/RD2)
  - 1 write data (in) port (A3/WD3, WE3 high enables writing)
  - 1 R15 Write Port (Always enabled)



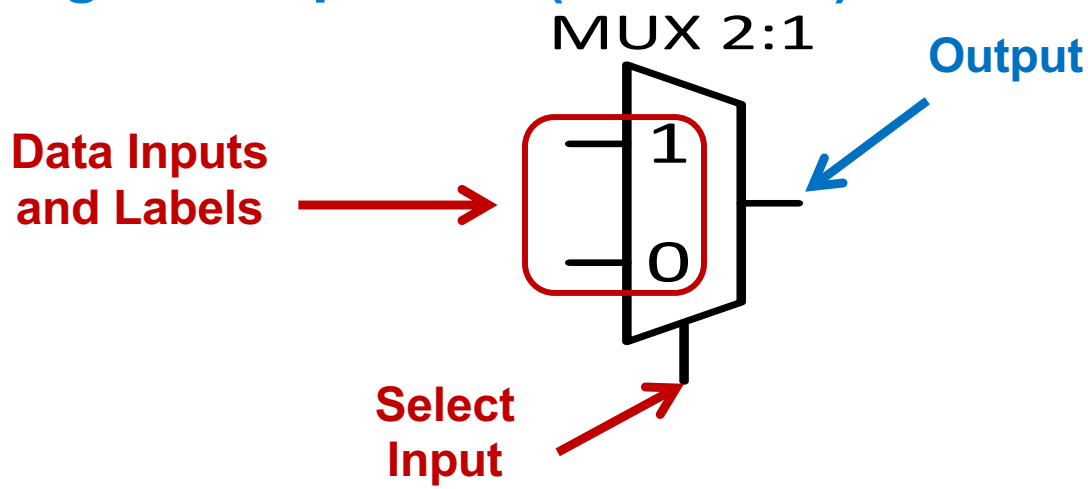
x

Clocked Circuit



- Clocked circuits employs a synchronizing signal called a **clock**
  - A clock is a periodic train of pulses; 0s and 1s
- A clock determines **when** computational activities occur
- Other inputs determine **what** changes will occur

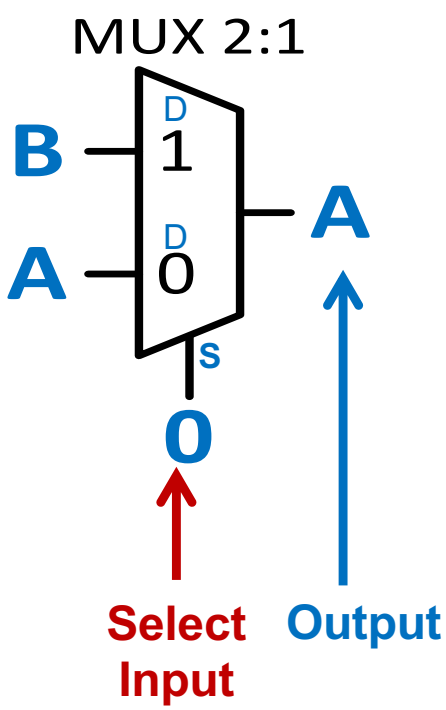
Steering Logic Multiplexers (or Muxes)

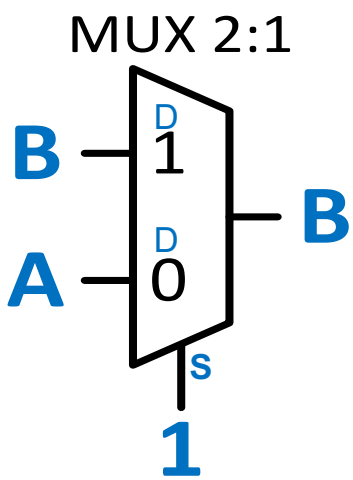


- **Connects *(routes)* one of 2^n inputs to the output**
  - 1 output
  - 2^n data inputs
  - n control lines selects just one of the inputs to route to the output
- "Selects" *(decides)* which input connects to output

x

Multiplexers (or Muxes)





2:1 MUX Functional

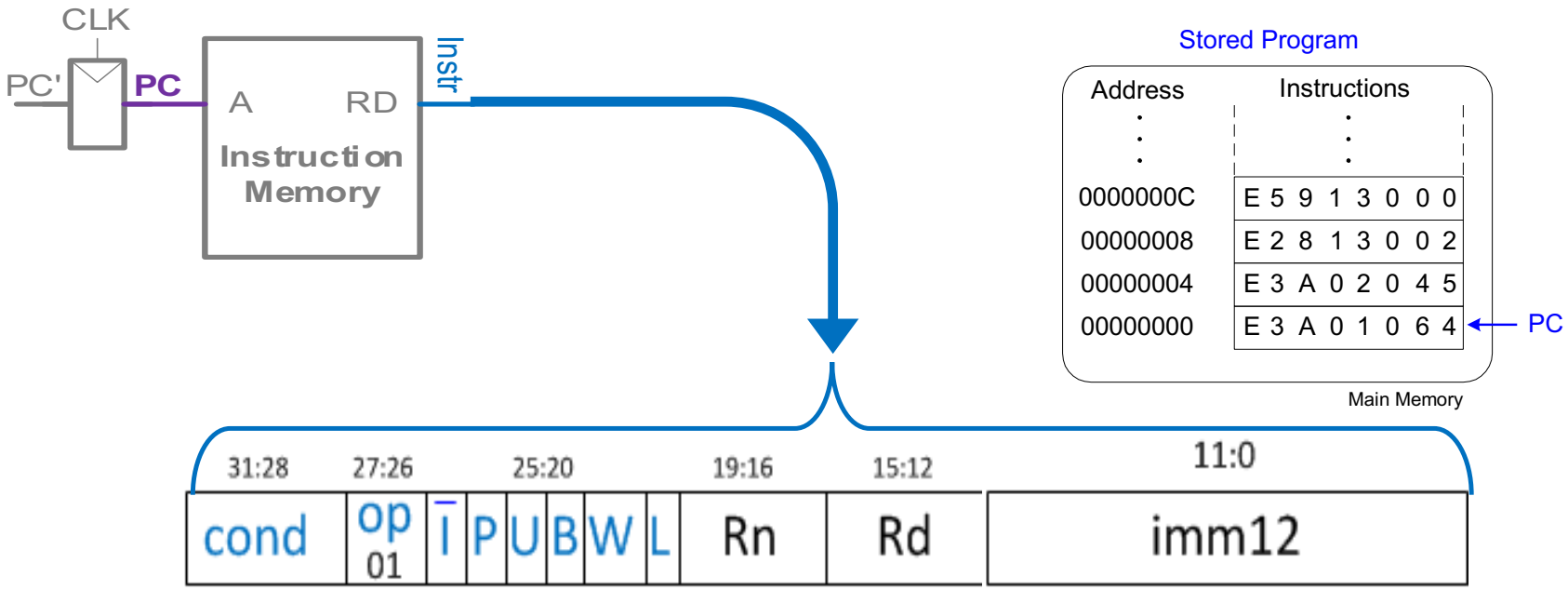
S	*Output*
0	A
1	B

x

Step 1: Instruction Fetch (ldr)

Steps through the program by fetching machine code from instruction memory

Assembly Code	Machine Code
`MOV R1, #100`	`0xE3A01064`
`MOV R2, #69`	`0xE3A02045`
`ADD R3, R1, R2`	`0xE2813002`
`STR R3, [R1]`	`0xE5913000`

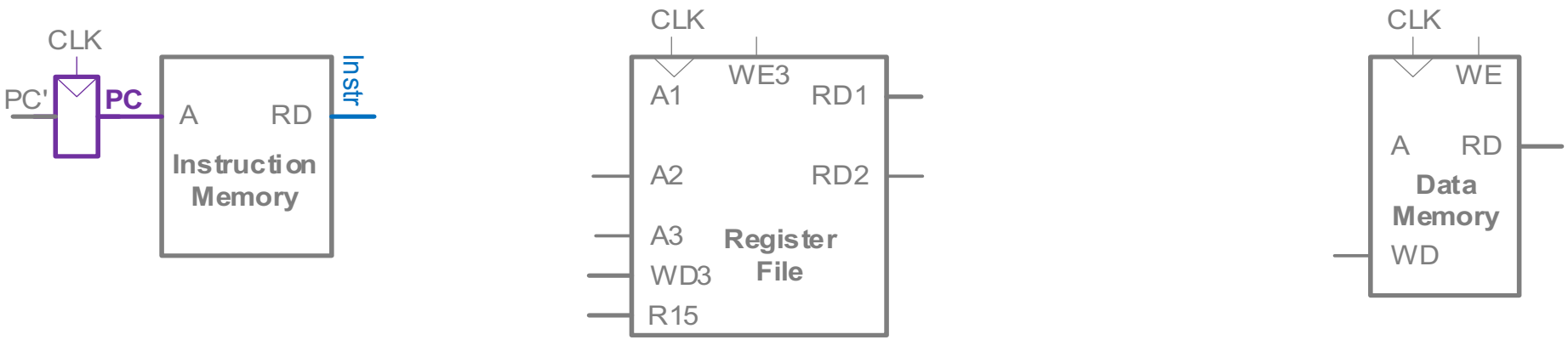


Hardware Decodes each instructions and uses logic + mux's on the bit fields to route the data to/from destinations and functional units

x

Step 1: Instruction Fetch (LDR)

```
ldr Rd, [Rn, imm12]
```

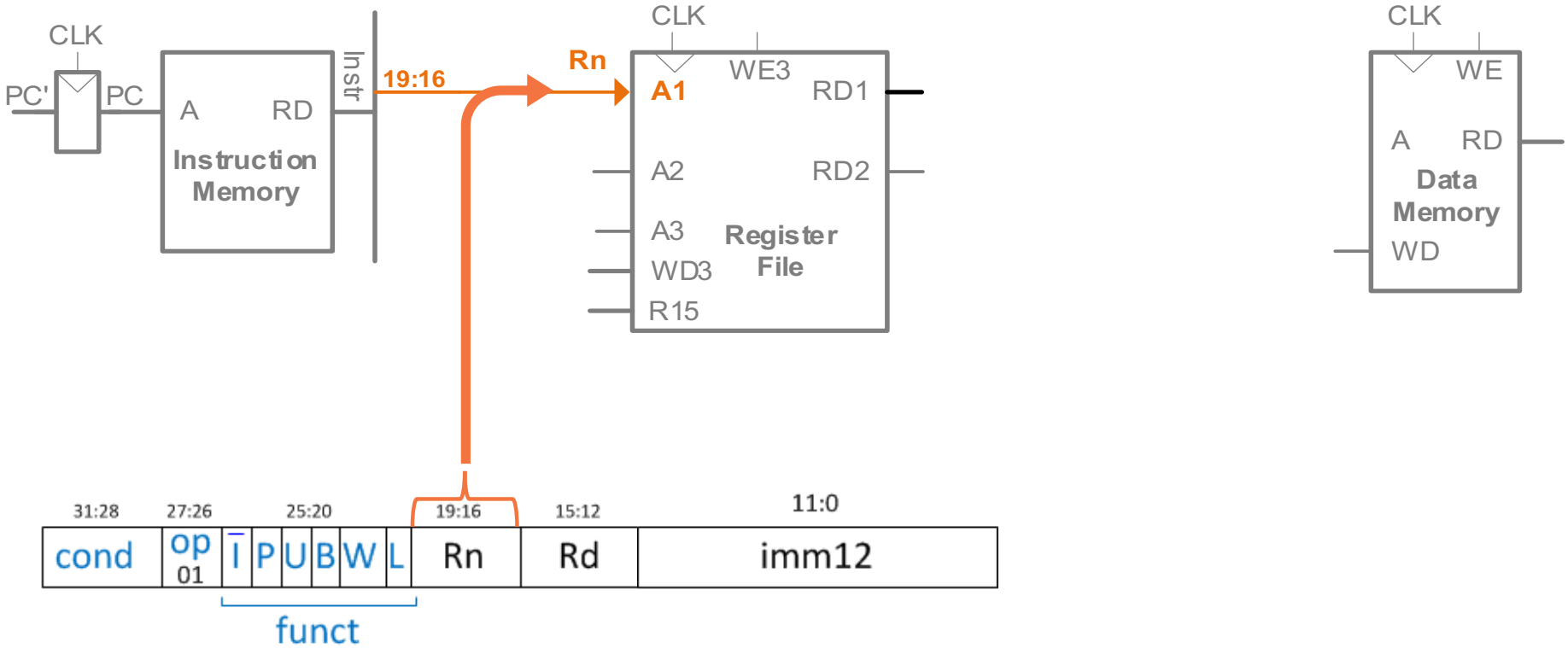


31:28	27:26	25:20						19:16	15:12	11:0
cond	op 01	$\overline{I}$	P	U	B	W	L	Rn	Rd	imm12

x

Step 2: Read source operands from Register File

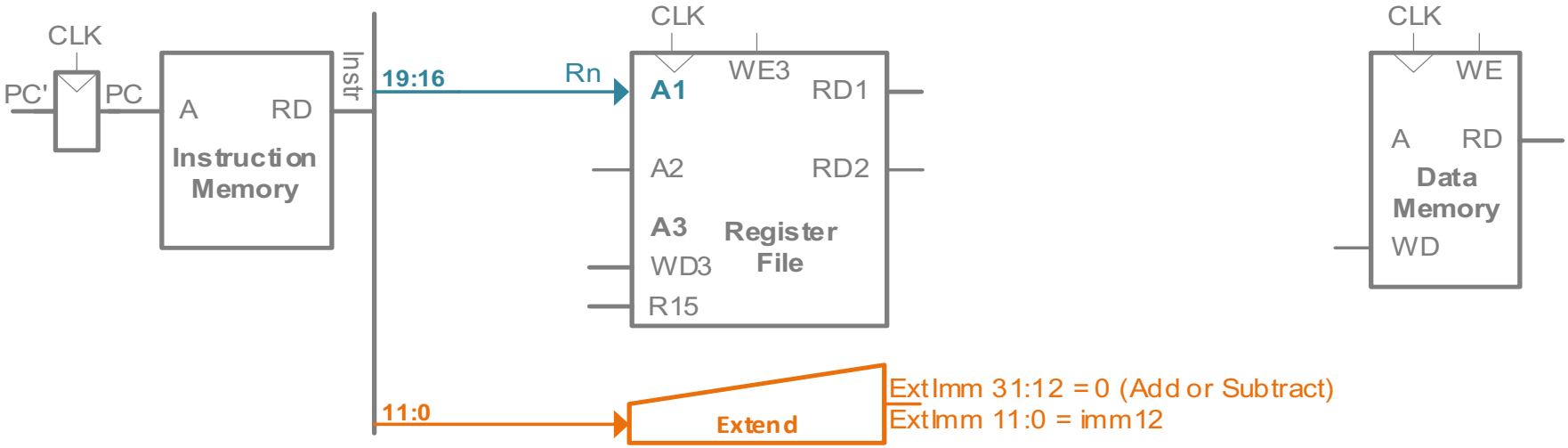
`ldr Rd, [Rn, imm12]`

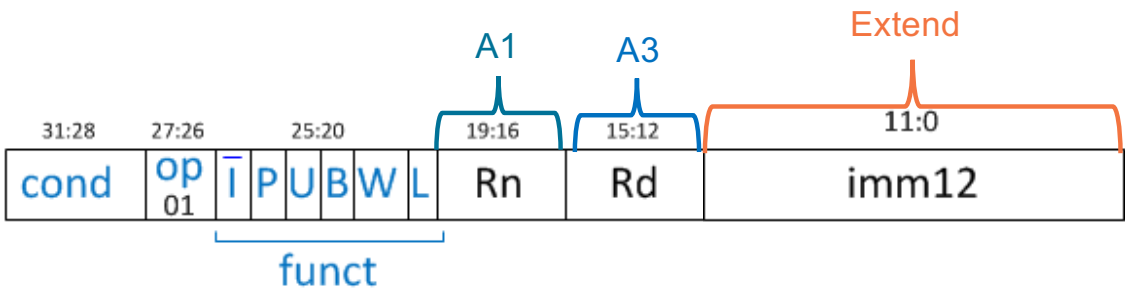


x

Step 3: Extend the Immediate Field From 12 to 32 bits

`ldr Rd, [Rn, imm12]`



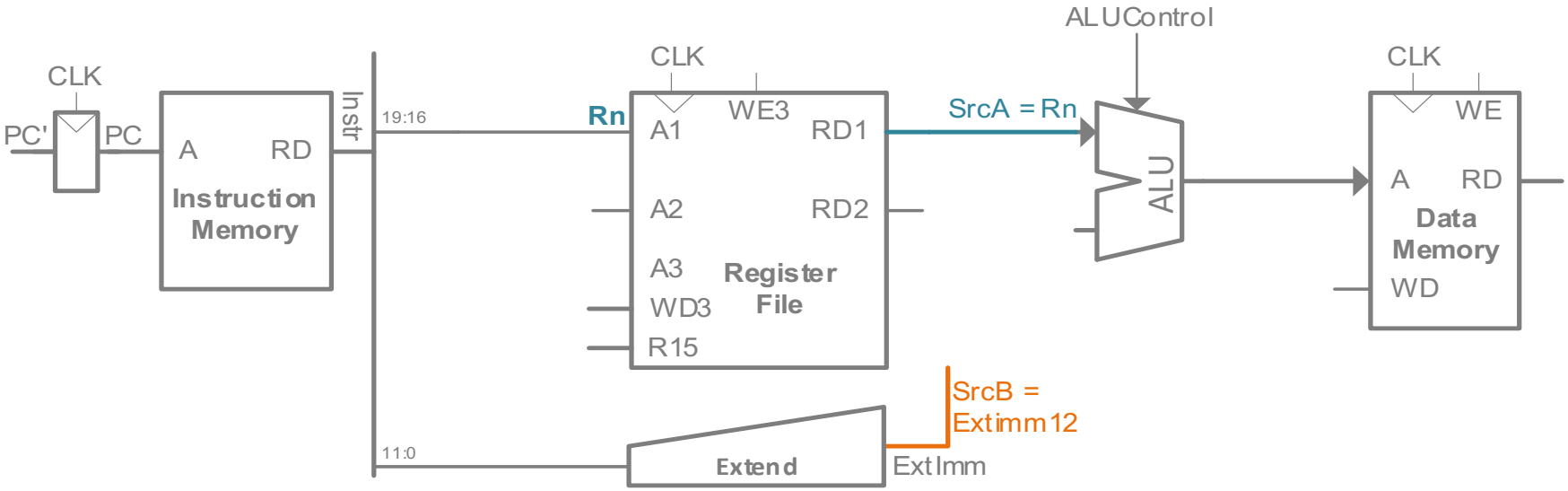


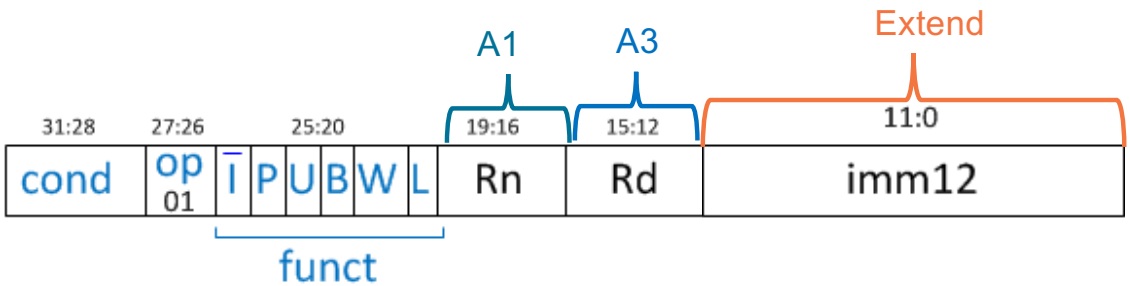
U	Operation
0	Sub From Base
1	Add to Base

x

Step 4: Read The Base Register

`ldr Rd, [Rn, imm12]`



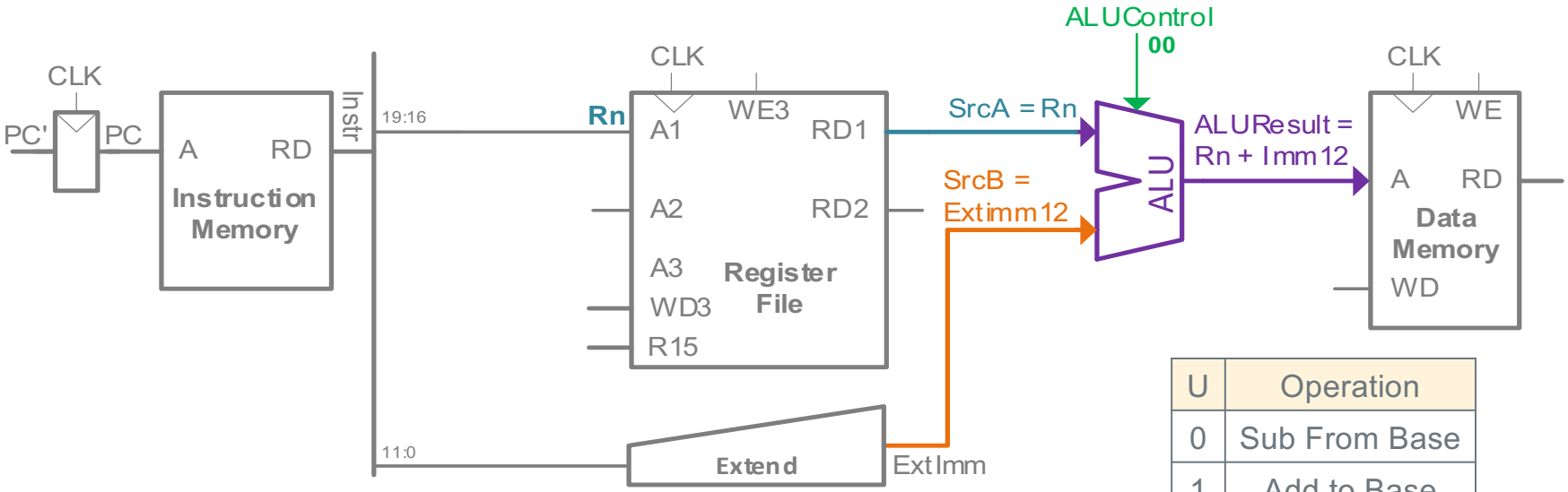


U	Operation
0	Sub From Base
1	Add to Base

x

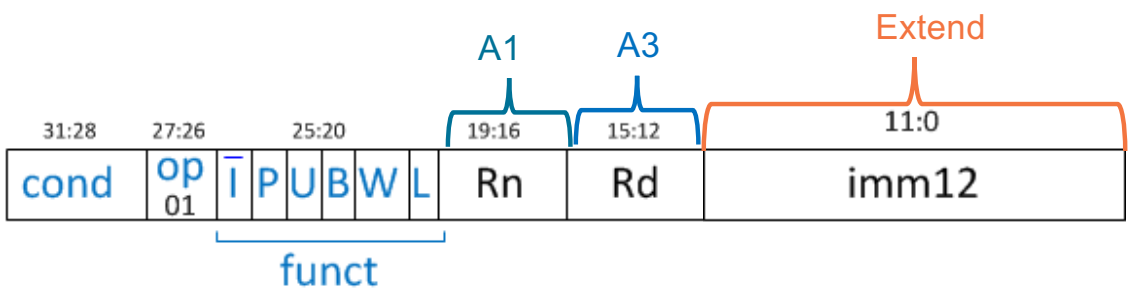
Step 5: Generate The memory address (Rn + imm12)

`ldr Rd, [Rn, imm12]`



U	Operation
0	Sub From Base
1	Add to Base

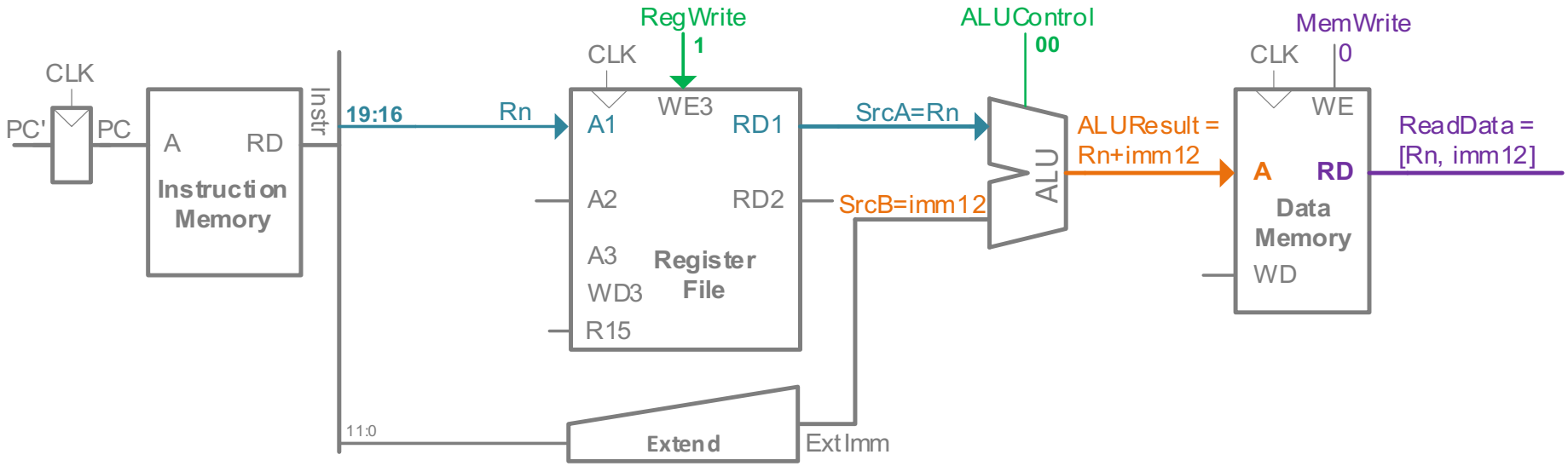
$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

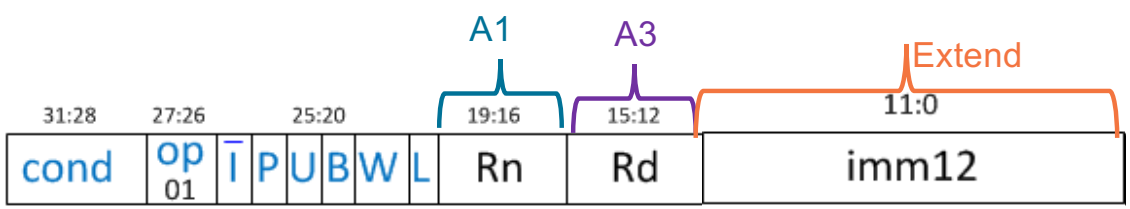


x

Step 6: Read Data from memory

`ldr Rd, [Rn, imm12]`



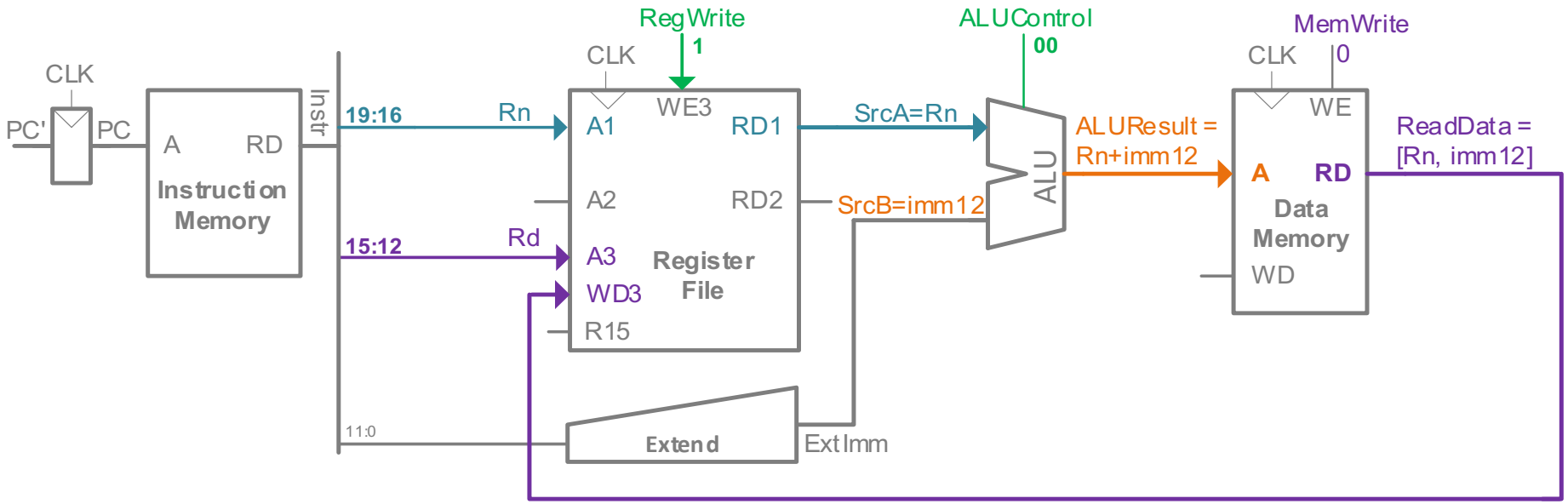


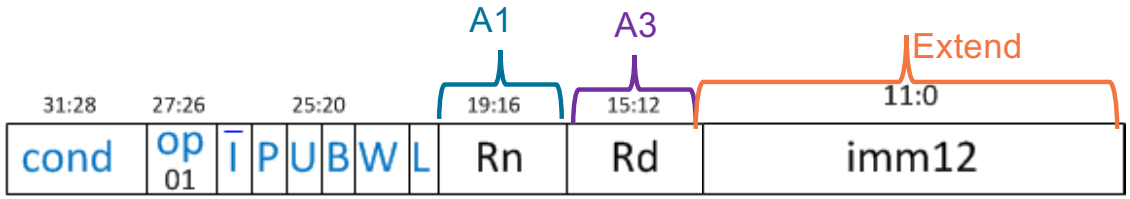
$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

x

Step 7: write Memory to Register Rd

`ldr Rd, [Rn, imm12]`

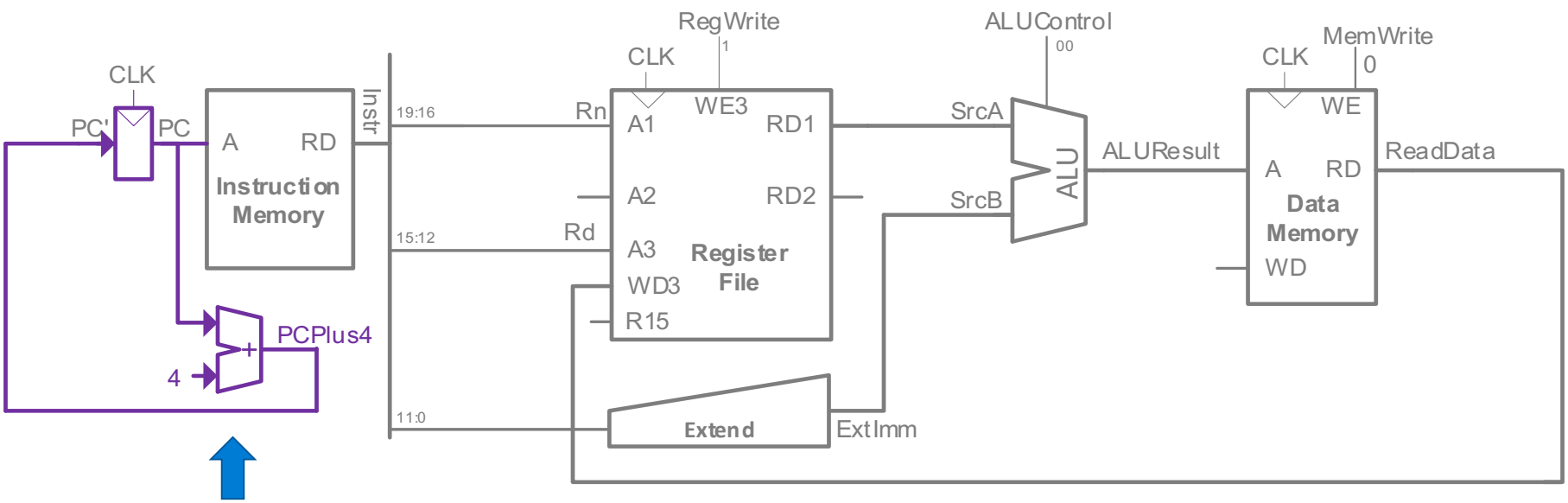




$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

x

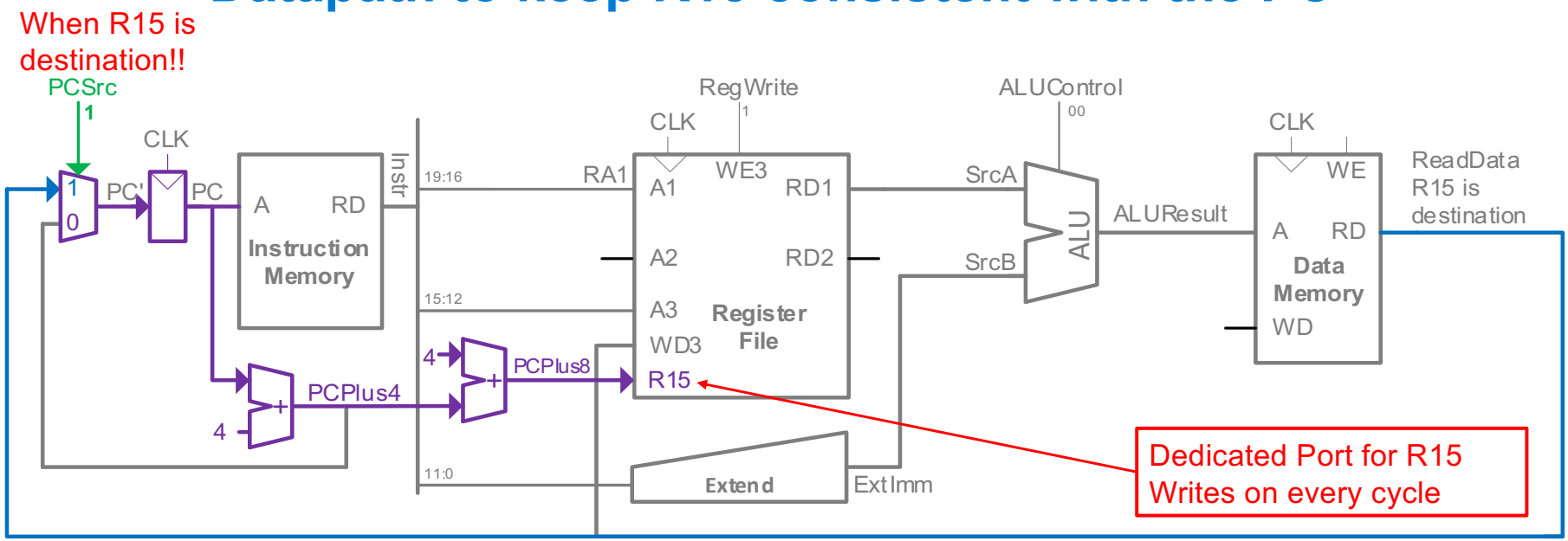
Step 8: Get The address of the next instruction



Dedicated + 4 adder for stepping through instructions

x

Datapath to keep R15 consistent with the PC

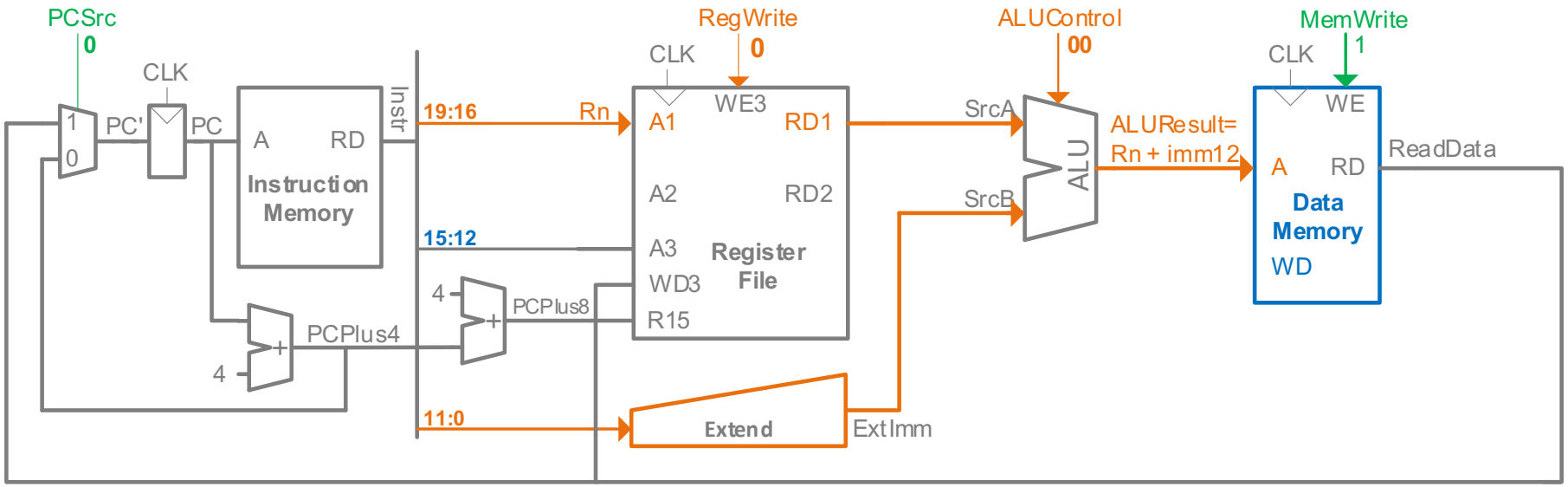


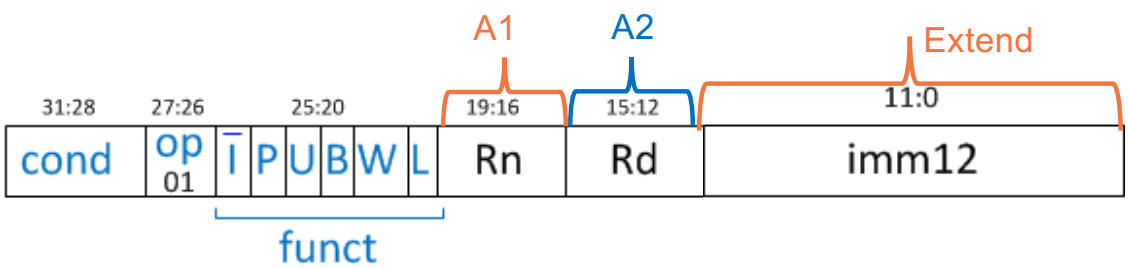
- **Source:** R15 must track the PC and be available to read/write in the Register File
  - **PC/r15** can be read explicitly in an instruction
  - Value in PC/r15 is = current **PC plus 8 (for pipelining – cse141) & <u>updated every cycle</u>!**
- **Destination:** Can write explicitly to PC/r15 (causes a branch as a side effect)

x

STR (Same as ldr except must write Rd to memory)

```
str Rd, [Rn, imm12]
```



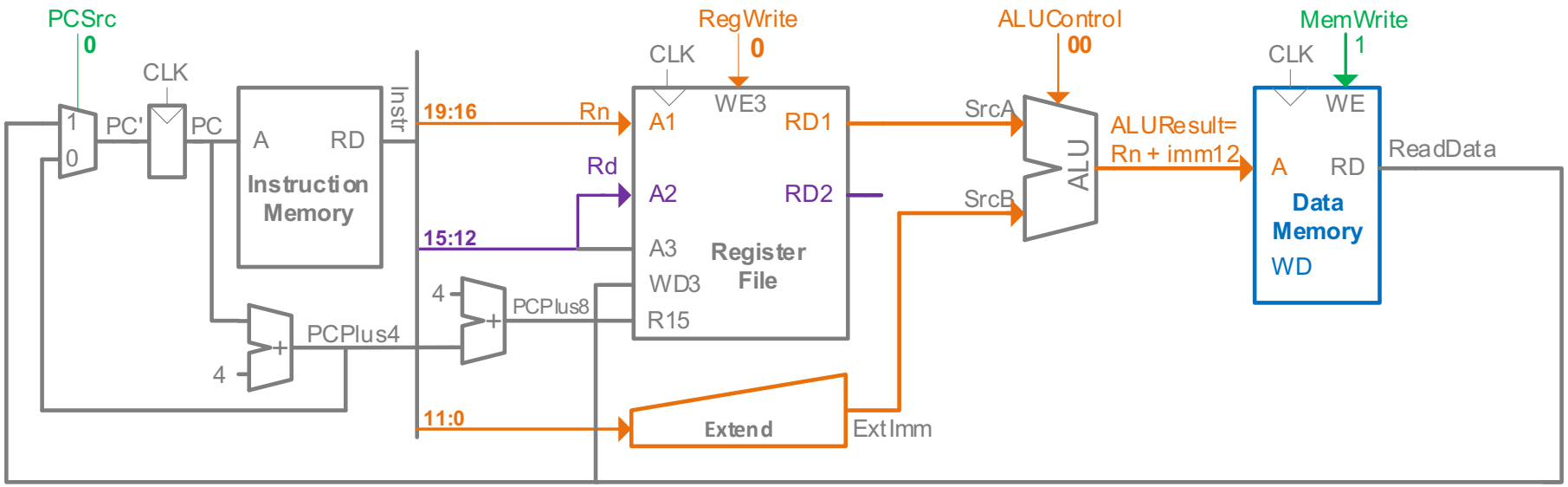


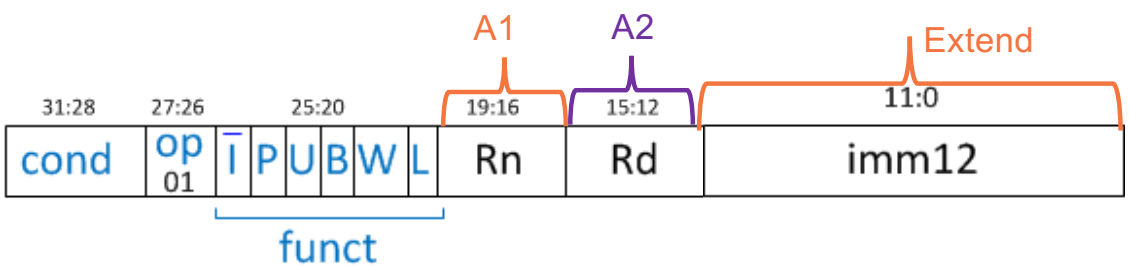
$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

x

STR Read Register Rd

`str Rd, [Rn, imm12]`



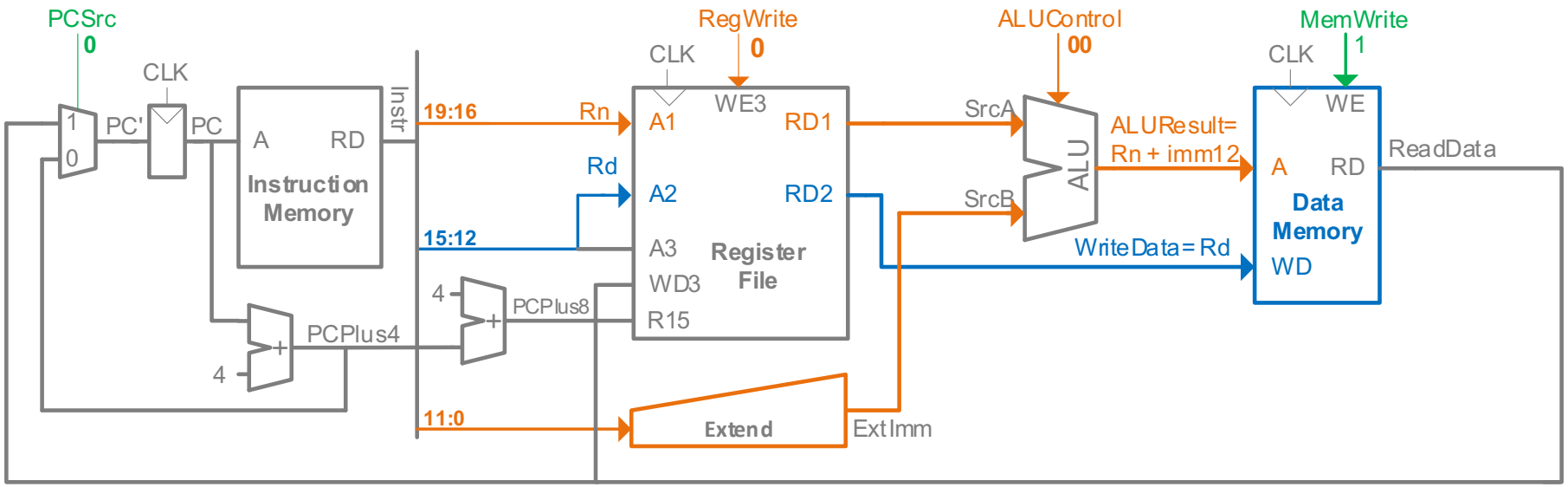


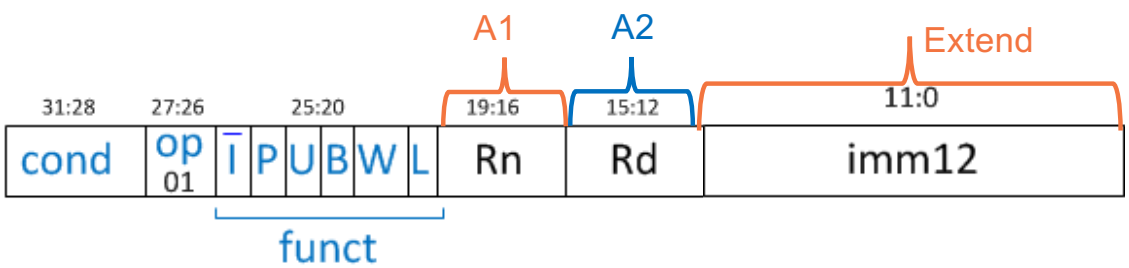
$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

x

STR write the contents of Rd to memory

`str Rd, [Rn, imm12]`





$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

x

Data Processing Instructions: Src2 immediate

Data-processing

31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
cond	op 00	I	cmd	S	Rn	Rd	rot	imm8

`add Rd, Rn, imm8`

- Must Modify extend logic
- Read from **`Rn`** and **`Imm8`** (*ImmSrc* chooses zero-extended `Imm8` instead of `Imm12`)
- Write *ALUResult* to register file
- Write to **`Rd`**

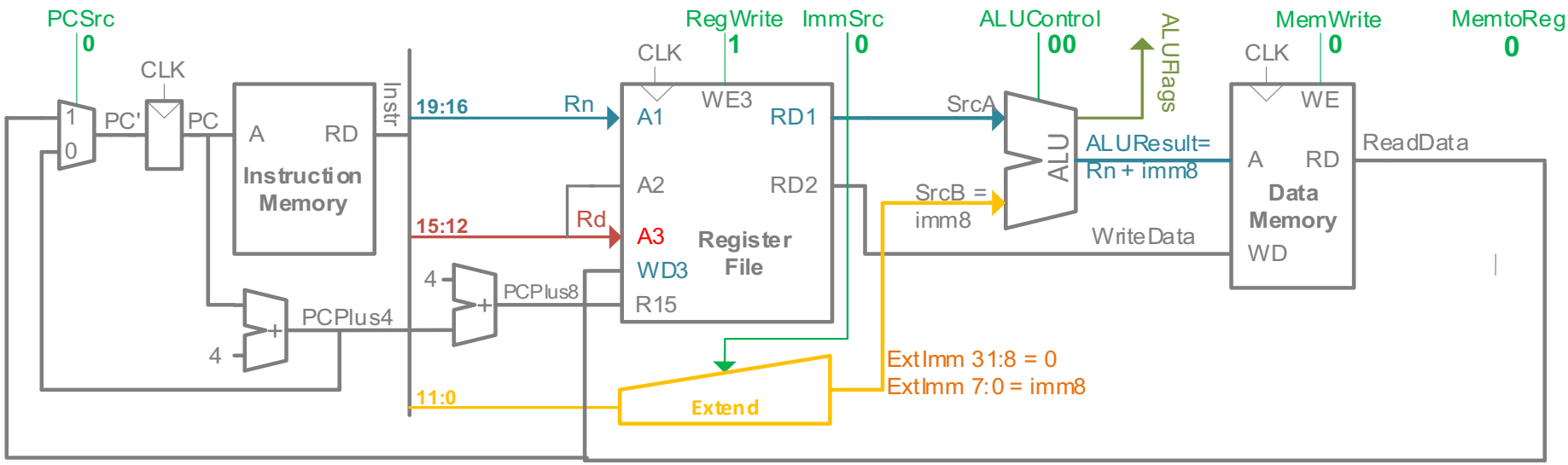
$ImmSrc_0$	ExtImm	Description
0	{24'b0, $Instr_{7:0}$}	Zero-extended *imm8*
1	{20'b0, $Instr_{11:0}$}	Zero-extended *imm12*

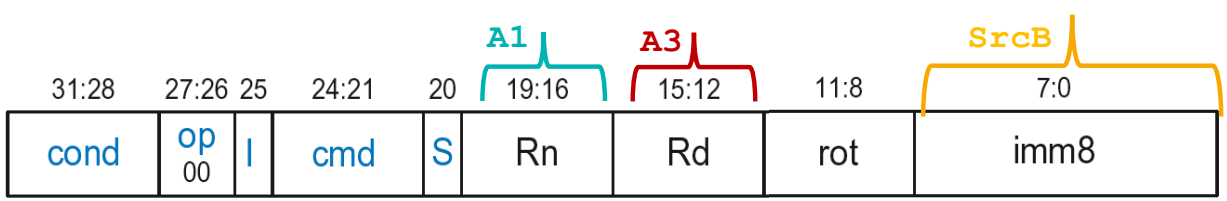
x

Data Processing Instructions: Src2 + immediate

`add Rd, Rn, imm8`

- Read from **`Rn`** **and** **`Imm8`** (control lines *ImmSrc* chooses the zero-extended `Imm8` instead of `Imm12`)
- We can get *ALUResult*, but how to route to the register file?





Control	Value
PCsrc	0
ReqWrite	1
ImmSrc	0
ALUcontrol	00
MemWrite	0
MemtoReg	0

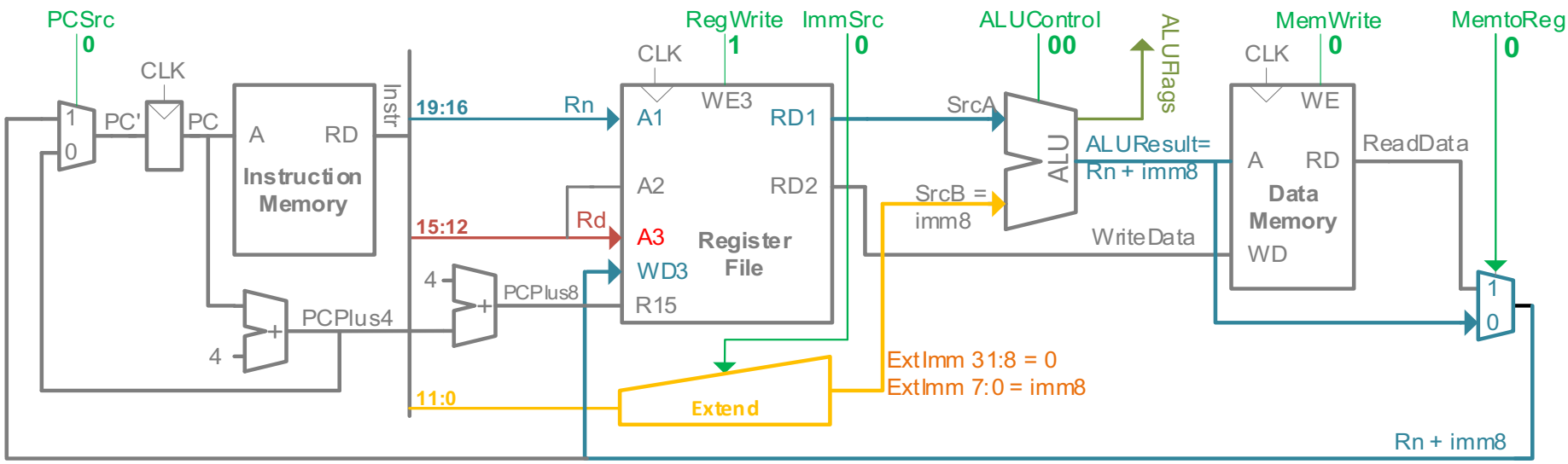
$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

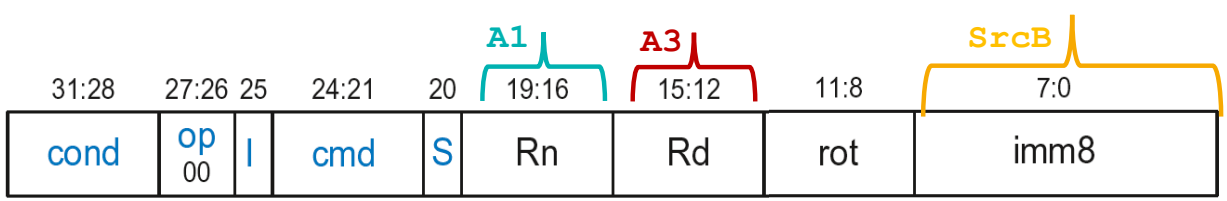
X

Data Processing Instructions: Src2 immediate

add Rd, Rn, imm8

- Solution: Add Mux to route ALU output around memory to Write to **Rd**





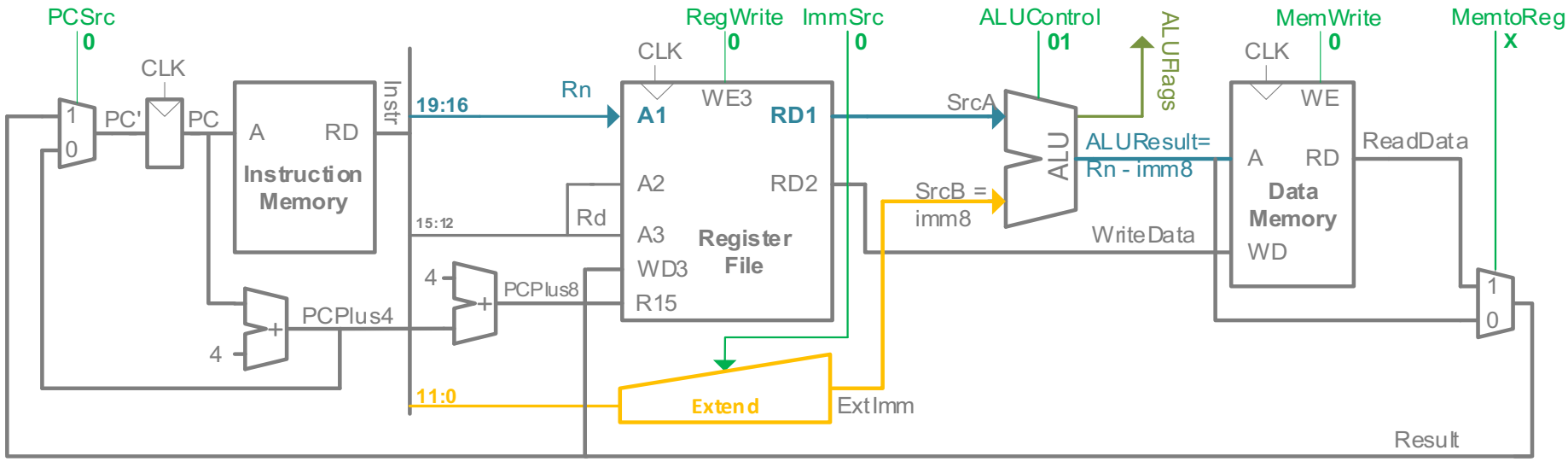
Control	Value
PCsrc	0
ReqWrite	1
ImmSrc	0
ALUcontrol	00
MemWrite	0
MemtoReg	0

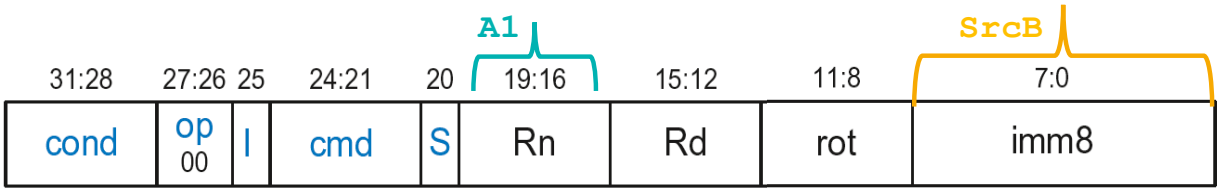
$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

X

Data Processing Instructions: Src2 immediate

- Read from **Rn and Imm8** (control lines *ImmSrc* chooses the zero-extended **Imm8** instead of **Imm12**)
- **Set ALUFlags, no Rd writeback!** `cmp Rn, imm8`





Control	Value
PCsrc	0
ReqWrite	0
ImmSrc	0
ALUcontrol	01
MemWrite	0
MemtoReg	x

$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

x

Data Processing Instructions: Src2 Register

- Read from **`Rn`** and **`Rm`** (instead of **`Imm8`**)
- Write *ALUResult* to register file
- Write to **`Rd`**

Data-processing

31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
cond	op 00	I	cmd	S	Rn	Rd	shamt5	sh	0	Rm

```
add Rd, Rn, Rm
```

$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

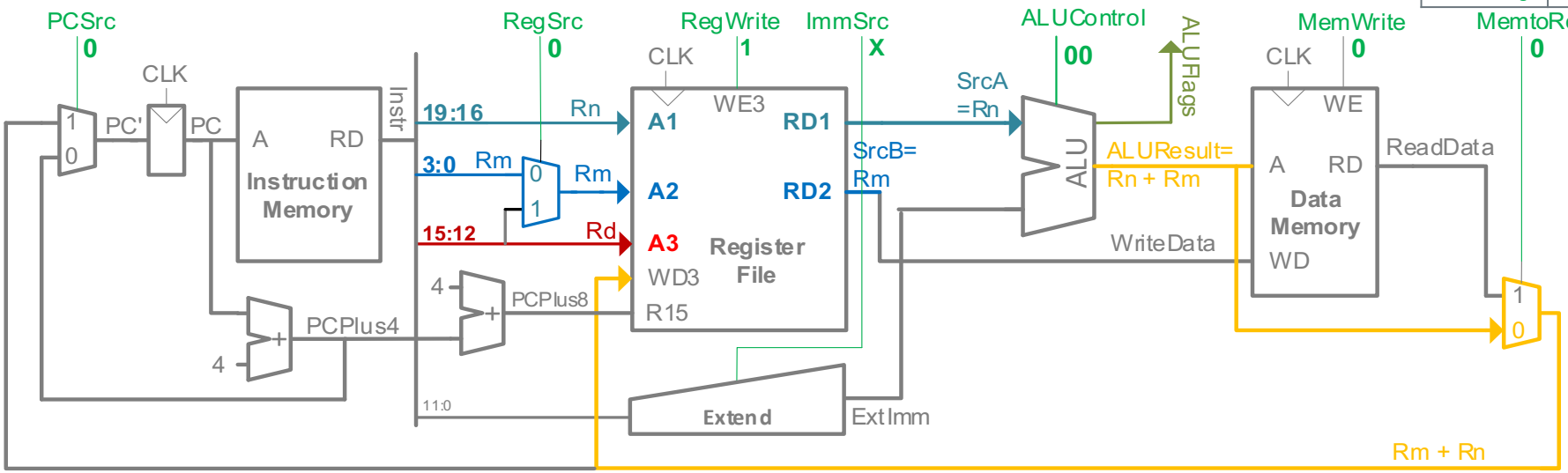
x

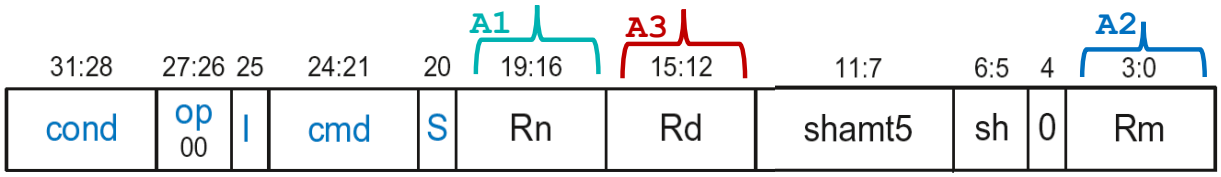
Data Processing Instructions: Src2 Register

- For RM we need to Add mux to route bits 3:0 to A2
- Read from **Rn** and **Rm** (instead of **Imm8**)
- Need to get RD2 to ALU!

Control	Value
PCsrc	0
RegSrc	0
ReqWrite	1
ImmSrc	X
ALUSrc	0
ALUcontrol	00
MemWrite	0
MemtoReg	0

add Rd, Rn, Rm





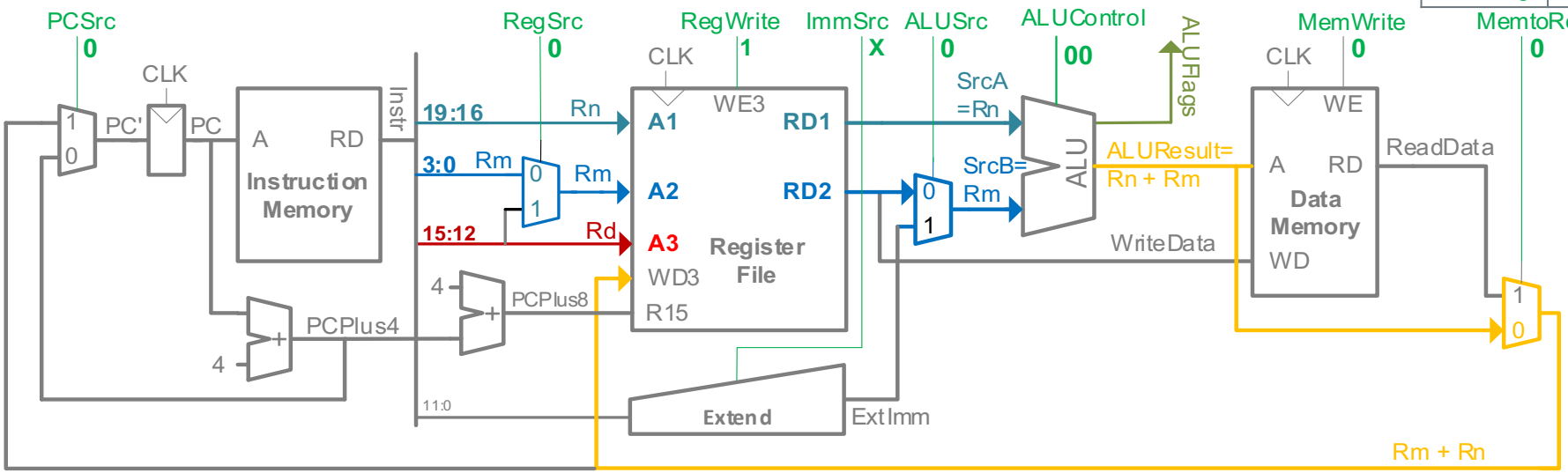
x

Data Processing Instructions: Src2 Register

- Add Mux to route RD2 to ALU, **and** another mux to route bits 3:0 to A2
- Read from **`Rn`** and **`Rm`** (instead of **`Imm8`**)
- Write *ALUResult* to register file
- Write to **`Rd`**

`add Rd, Rn, Rm`

Control	Value
PCsrc	0
RegSrc	0
ReqWrite	1
ImmSrc	X
ALUSrc	0
ALUcontrol	00
MemWrite	0
MemtoReg	0



31:28	27:26	25	24:21	20	19:16 (A1)	15:12 (A3)	11:7	6:5	4	3:0 (A2)
cond	op 00	I	cmd	S	Rn	Rd	shamt5	sh	0	Rm

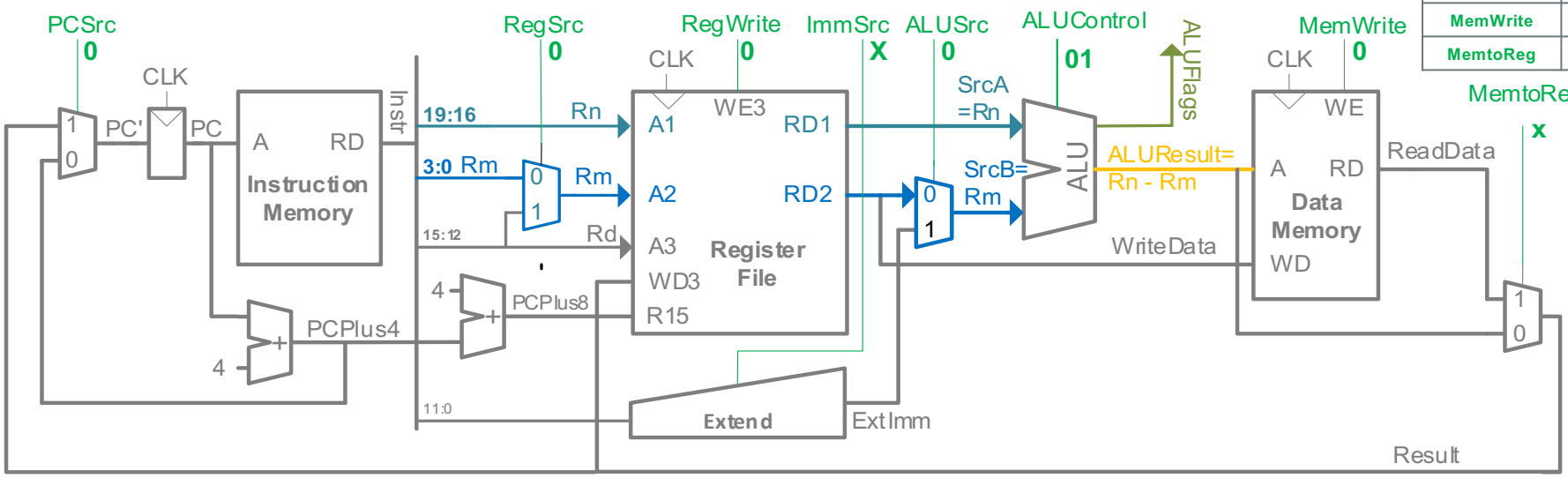
x

Data Processing Instructions: Src2 Register

- Read from **Rn** and **Rm** (instead of **Imm8**)
- Set ALUFlags

cmp Rn, Rm

Control	Value
PCsrc	0
RegSrc	0
ReqWrite	0
ImmSrc	X
ALUSrc	0
ALUcontrol	01
MemWrite	0
MemtoReg	0



31:28	27:26	25	24:21	20	19:16 (A1)	15:12	11:7	6:5	4	3:0 (A2)
cond	op 00	I	cmd	S	Rn	Rd	shamt5	sh	0	Rm

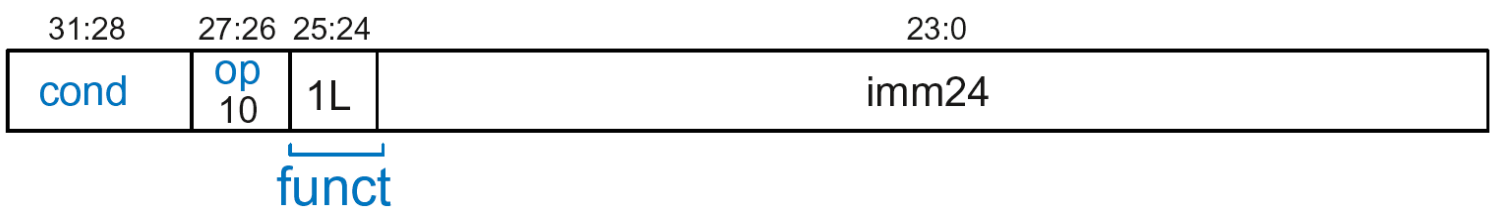
x

Branch Instruction Format (B and BL)

- ***Branch Target Address*** **(BTA)**
  - Next PC when branch taken
  - BTA is relative to current PC + 8
- ***imm24*** = # of words (instructions) BTA is away from PC+8

```
0xA0            BLT THERE      ← PC
0xA4            add r0, r1, r2
0xA8            sub r0, r0, r9 ← PC+8
0xAC            add sp, sp, 8
0xB0            bx lr
0xB4 THERE      sub r0, r0, 1  ← BTA
0xB8            bl  TEST
```

- PC = 0xA0
- PC + 8 = 0xA8
- **THERE** label is 3 instructions past PC+8
- So, *imm24* = 3



x

Branch Instruction: B

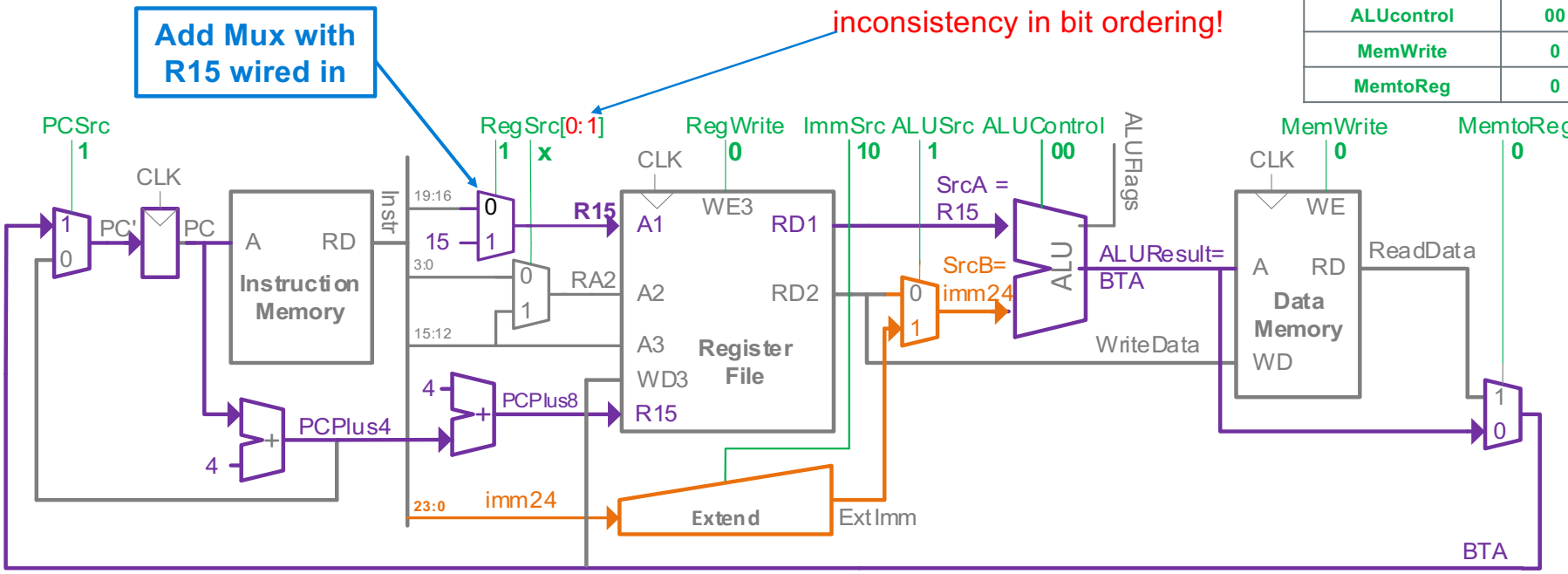
Calculate branch target address:

ExtImm = Imm24 << 2 and sign-extended

BTA = (*ExtImm*) + (PC + 8)

Control	Value
PCsrc	1
RegSrc[1:0]	X1
ReqWrite	0
ImmSrc	10
ALUSrc	1
ALUcontrol	00
MemWrite	0
MemtoReg	0

Be careful of textbook inconsistency in bit ordering!



31:28	27:26	25:24	23:0
cond	op 10	1L	imm24

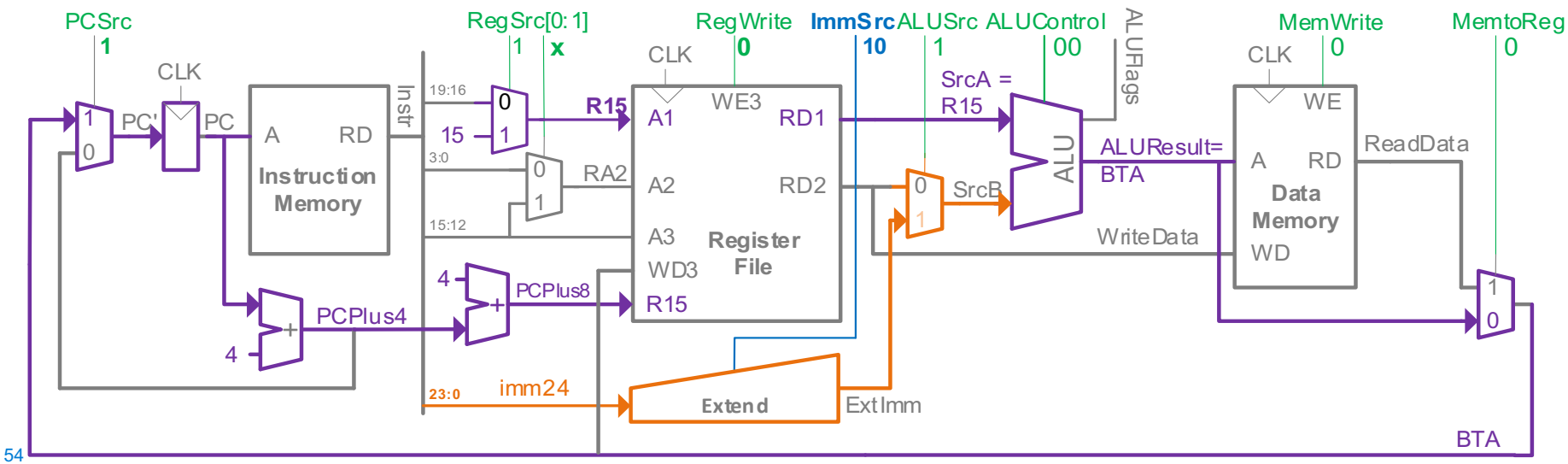
B Label

x

Branch Instruction: Extend Functional Unit

$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

$ImmSrc_{1:0}$	ExtImm	Description
00	{24'b0, $Instr_{7:0}$}	Zero-extended *imm8*
01	{20'b0, $Instr_{11:0}$}	Zero-extended *imm12*
10	{6{$Instr_{23}$}, $Instr_{23:0}$} 00	Sign-extended *imm24*



x

Control	Add r2,r1, 4 Rd, Rn, imm8	cmp r1, 4 Rn, imm8	Add r3,r7,r1 Rd, Rn, Rm	sub r3,r7,r1 Rd, Rn, Rm	and r3,r7,r1 Rd, Rn, Rm	Str r5, [r6, #44] Rd, [Rn,imm12]	ldr r4, [r0, r1] Rd, [Rn, Rm]	b label s_ext_imm24
PCsrc	0	0	0	0	0	0	0	1
RegSrc[1:0]	X0	X0	00	00	00	10	00	X1
ReqWrite	1	0	1	1	1	0	1	0
(n) RA1[3:0]	0001	0001	0111	0111	0111	0110	0000	1111
(m) RA2[3:0]	xxxx	xxxx	0001	0001	0001	0101	0001	xxxx
(d) RA3[3:0]	0010	xxxx	0011	0011	0011	XXXX	0100	xxxx
ImmSrc[1:0]	00	00	xx	xx	xx	01	xx	10
ALUSrc	1	1	0	0	0	1	0	1
ALUcontrol[1:0]	00	01	00	01	10	00	00	00
MemWrite	0	0	0	0	0	1	0	0
MemtoReg	0	0	0	0	0	X	1	0