3 : The Processor (Single Cycle)

Acknowledgement :

- Text and companion slides by Harris and Harris
- Text by Patterson and Hennessey and companion slides by Mary Jane Irwin

Note:

- Not all slides will be covered in the lecture. The rest are left as a self-learning exercise.

Contents

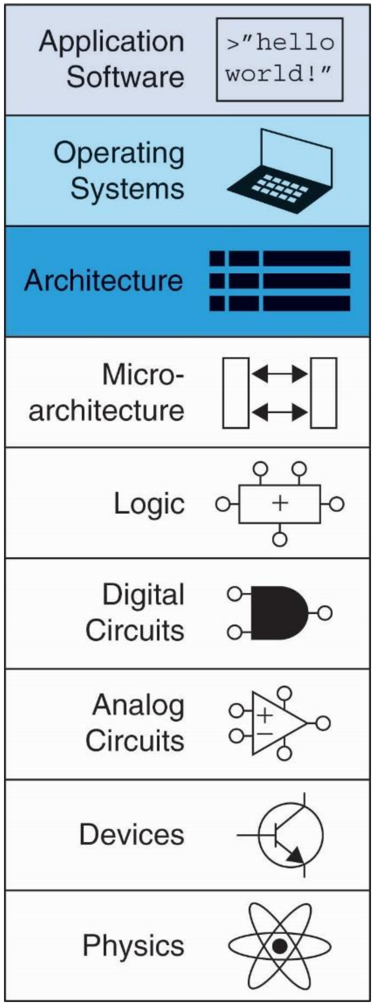
- ARM architecture overview
- ARM7 (ARMv3) instruction encoding
- Microarchitecture : Datapath design
- Microarchitecture : Controller design
- Extensions for other formats / instructions

Recap : Architecture vs Microarchitecture

- Architecture: programmer's view of computer
  - Defined by instructions & operand locations
  - Assembly language: human-readable format of instructions
  - Machine language: computer-readable format (1's and 0's)
  - Assembly language -> Machine language conversion is done by the assembler
    - one to one correspondence

      (except for pseudo-instructions)

- Microarchitecture: how to implement an architecture in hardware



Why Learn ARM?

- Developed in the 1980's by Advanced RISC Machines – now called ARM Holdings
- More than 10 billion ARM processors sold/year
- Almost all cell phones and tablets have multiple ARM processors
- Well over 75% of humans use products with an ARM processor
- Used in servers, cameras, robots, cars, pinball machines,… etc.
- ARM provides
  - architecture licenses
    - Companies design their own microarchitectures [and SoCs using these microarchitectures]
    - e.g., Qualcomm Kryo [Snapdragon SoC], Apple Avalanche, Blizzard [A15 SoC]

    as well as

  - microarchitecture licenses
    - Companies use microarchitectures from ARM [and SoCs using these microarchitectures]
    - e.g., Cortex M3 [LPC 1769 SoC], Cortex-A710 [MediaTek Helio Dimensity 9000 SoC]

ARM Families and Architectures

- Architectures evolve. Complete list of families and architectures at https://en.wikipedia.org/wiki/List_of_ARM_microarchitectures
- CG2028/EE2028/EE2024 was based on ARMv7-M* ISA
  - Newer (beware of legal issues if you try to implement)
  - Executes Thumb (16-bit instructions, 32 bit data) and Thumb 2 (32-bit)
  - Thumb – an encoding which allows better code density (with some limitations such as limited number of usable registers)
  - Thumb 2 – 32 bit instructions to extend Thumb functionality
  - ARMv7-M = Microcontroller; ARMv7-A = Application (phones etc.)
- In this module, we will be building an ARMv3 ISA based processor
  - Original 32-bit instruction set, implemented in ARM7 microarchitecture
  - Older (~1994, so you are unlikely to run into legal issues. DO NOT put it up on public repos though, to be on the safe side)
  - Somewhat similar to ARMv7-M at assembly level, though very different at machine level

*LPC1769 from NXP is an SoC based on ARM Cortex M3 microarchitecture which implements ARMv7-M ISA

ARM / RISC Design Principles

- Regularity supports design simplicity
  - Constant length instructions (and data) and consistent instruction format
  - Same number of operands (two sources and one destination)
  - Ease of encoding and handling in hardware
- Make the common case fast
  - ARM includes only simple, commonly used instructions
  - Hardware to decode and execute instructions kept simple, small, and fast
  - More complex instructions (that are less common) performed using multiple simple instructions
- Smaller is faster
  - ARM includes only a small number of registers
- Good design demands good compromises
  - Number of instruction formats kept small. Multiple instruction formats allow flexibility… but complicates hardware

Architecture : Registers, Instruction Formats

- ARM has 16 general purpose registers, with some special functions or typical uses
- We will be looking at 3 instruction formats
  - Data-processing
    - ADD, SUB, CMP, AND, ORR, MOV (LSL, ROR etc. are variants of MOV)
  - Memory
    - LDR, STR, LDRB, STRB etc.
  - Branch
    - B, BL

Load Byte. To load a *Byte* from the location (LDR loads a *Word*)

Register Name	Function / Typical Use
R0	Argument / return value / temporary variable
R1-R3	Argument / temporary variables
R4-R11	Saved variables
R12	Temporary variable
R13 (SP)	Stack Pointer
R14 (LR)	Link Register
R15 (PC)	Program Counter

Architecture : Conditional Execution

- Every instruction is conditionally executed based on N, Z, C, V flags. Think of flags as 1-bit registers (actually, they are bits within xPSR)
  - Example : ADDEQ R1, R2, R3 affects the value of R1 only when Z = 1

- Flags are set by instructions with suffix S
  - Example : ADDS affects flags, ADD doesn’t
  - Exceptions : CMP, CMN, TST, TEQ which are used only to set flags (result is discarded)

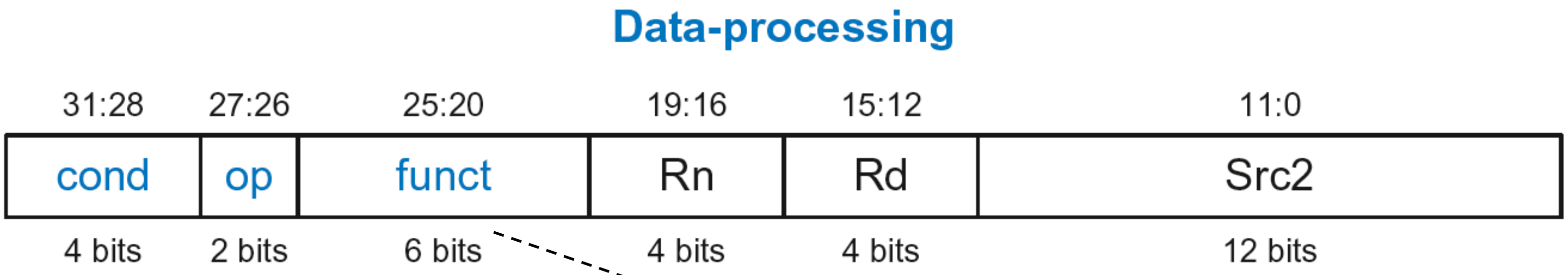
cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	$\bar{Z}$
0010	CS / HS	Carry set / Unsigned higher or same	C
0011	CC / LO	Carry clear / Unsigned lower	$\bar{C}$
0100	MI	Minus / Negative	N
0101	PL	Plus / Positive of zero	$\bar{N}$
0110	VS	Overflow / Overflow set	V
0111	VC	No overflow / Overflow clear	$\bar{V}$
1000	HI	Unsigned higher	$\bar{Z}C$
1001	LS	Unsigned lower or same	$Z\ OR\ \bar{C}$
1010	GE	Signed greater than or equal	$\overline{N \oplus V}$
1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\bar{Z}(\overline{N \oplus V})$
1101	LE	Signed less than or equal	$Z\ OR\ (N \oplus V)$
1110	AL (or none)	Always / unconditional	ignored

Interpretation based on SUBS/CMP

Data-processing Instruction Format

<OP Rd, Rn, Src2>
OP => ADD, SUB, MOV,..

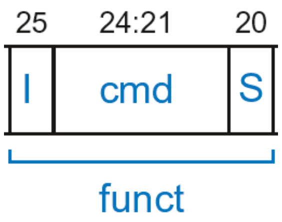
- Operands
  - *Rn* : first source register
  - *Src2* : second source – register or immediate
  - *Rd* : destination register



Data-processing

31:28	27:26	25:20	19:16	15:12	11:0
cond	op	funct	Rn	Rd	Src2
4 bits	2 bits	6 bits	4 bits	4 bits	12 bits

- Control fields
  - cond : specifies conditional execution
  - op : the operation code or opcode
    - op = "00" for data-processing (DP) instructions
  - funct : the function/operation to perform
    - funct is composed of cmd, I-bit, and S-bit



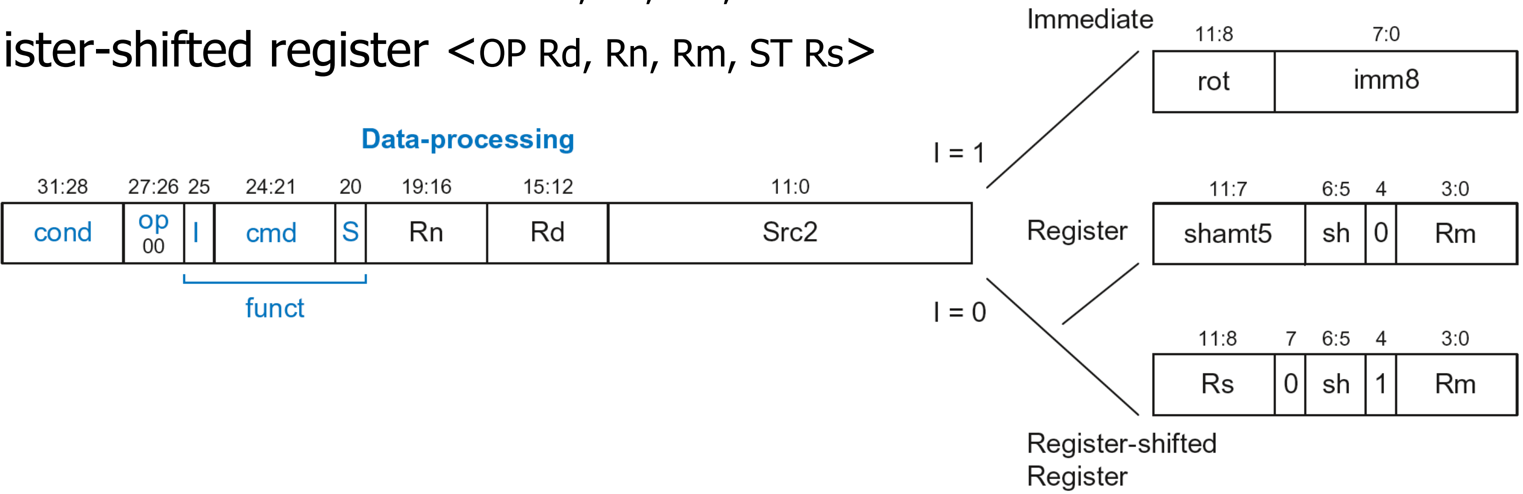
25	24:21	20
I	cmd	S

funct

Data-processing Instruction Format

- cmd : specifies the specific data-processing instruction <OP Rd, Rn, Src2>
  - cmd = "0100" for ADD; cmd = "0010" for SUB; ...
  - See page A3-9 of the ARM Architecture Reference Manual for the complete cmd code list
- I-bit
  - I = 0: *Src2* is a register. I = 1: *Src2* is an immediate
- S-bit : 1 if instruction sets condition flags
  - S = 0: SUB R0, R5, R7. S = 1: ADDS R8, R2, R4 or CMP R3, #10
- *Src2* can be
  - Immediate <OP Rd, Rn, #Imm8_rot>
  - Register with imm. shift <OP Rd, Rn, Rm, ST #shamt5>
  - Register-shifted register <OP Rd, Rn, Rm, ST Rs>

ST = Shift Type = LSL / LSR / ASR / ROR



Data-processing

31:28	27:26	25	24:21	20	19:16	15:12	11:0
cond	op 00	I	cmd	S	Rn	Rd	Src2

Immediate

11:8	7:0
rot	imm8

Register

11:7	6:5	4	3:0
shamt5	sh	0	Rm

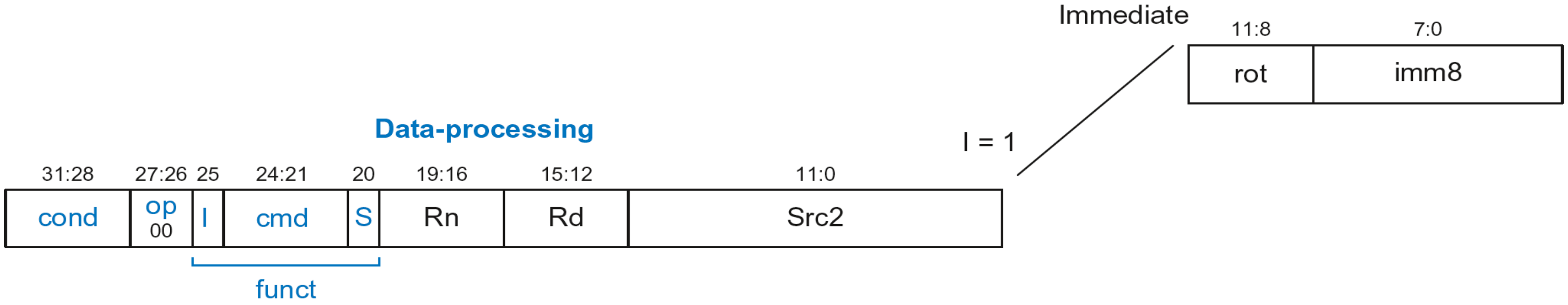
Register-shifted Register

11:8	7	6:5	4	3:0
Rs	0	sh	1	Rm

DP Instruction with Immediate *Src2*

<OP Rd, Rn, #Imm8_rot>

- Immediate encoded as
  - *Imm8* : 8-bit unsigned immediate
  - *rot* : 4-bit rotation value
- 32-bit constant is: *Imm8* ROR (*rot* × 2)



The assembly language programmer need not break down the 32-bit constant into imm8 and rot - the assembler will do this. If the 32-bit constant is such that it can't be encoded using imm8 and rot, the assembler will flag an error. For constants that fit within 8 bits, rot = 0.

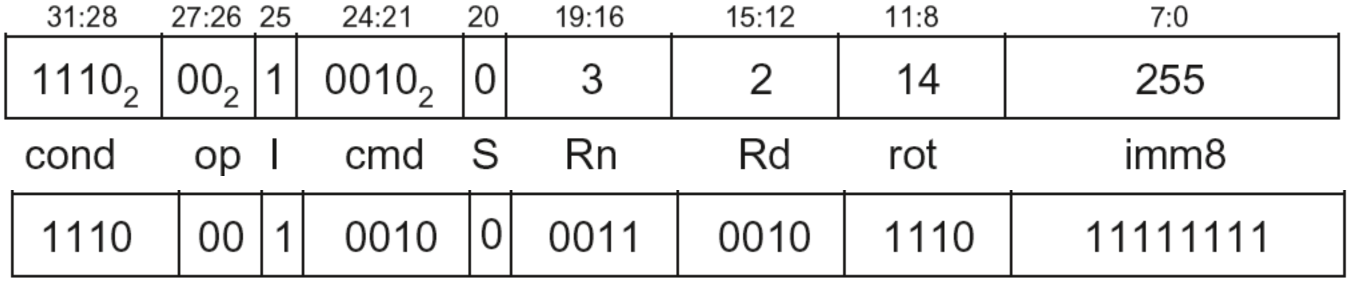
Note: This is very different from ARM v7-M / Thumb where *Src2* / *Operand2* is Imm8m, formed by left-shifting an 8-bit value by any number of bits, or a bit pattern of one of the forms 0xXYXYXYXY, 0x00XY00XY or 0xXY00XY00

DP Instruction with Immediate *Src2* : e.g.

<OP Rd, Rn, #Imm8_rot>

- SUB R2, R3, #0xFF0
  - cond = “1110” (14) for unconditional execution
  - op = “00” (0) for data-processing instructions
  - cmd = “0010” (2) for SUB
  - *Src2* is an immediate so I=1
  - *Rd* = 2, *Rn* = 3
  - *Imm8* = 0xFF
  - *Imm8* must be rotated right by 28 to produce 0xFF0, so *rot* = 14

ROL by 4 => ROR by (32-4) = 28
rot × 2 = 28 => *rot* = 14



31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
1110_2	00_2	1	0010_2	0	3	2	14	255
cond	op	I	cmd	S	Rn	Rd	rot	imm8
1110	00	1	0010	0	0011	0010	1110	11111111

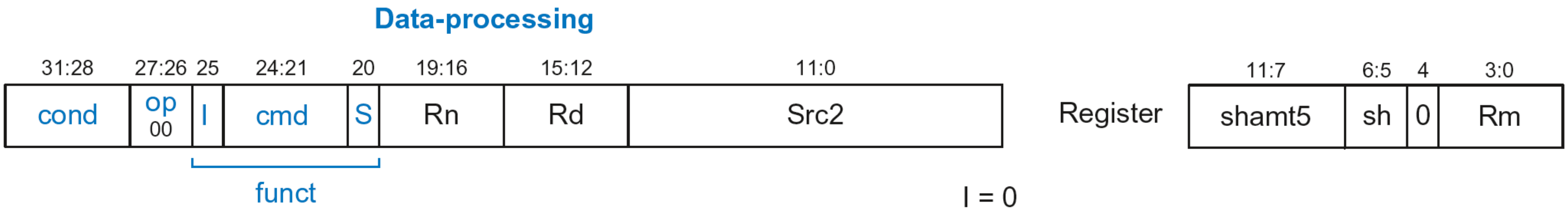
SUB R2, R3, #0xFF0 => 0xE2432EFF

DP Instruction with Register *Src2*

<OP Rd, Rn, Rm, ST #shamt5>

- *Rm* : the second source operand
- *shamt5* : the amount Rm is shifted
- *sh* : the type of shift

Shift Type	*sh*
LSL	00
LSR	01
ASR	10
ROR	11



Data-processing

31:28	27:26	25	24:21	20	19:16	15:12	11:0
cond	op 00	I	cmd	S	Rn	Rd	Src2

funct

Register

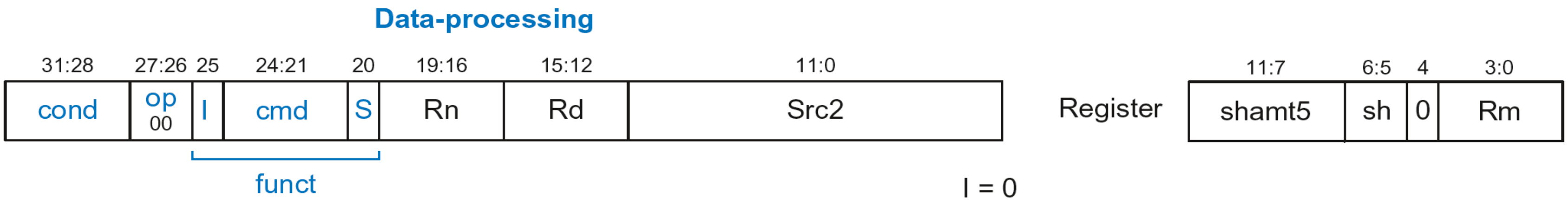
11:7	6:5	4	3:0
shamt5	sh	0	Rm

I = 0

DP Instruction with Register *Src2* : e.g.

<OP Rd, Rn, Rm, ST #shamt5>

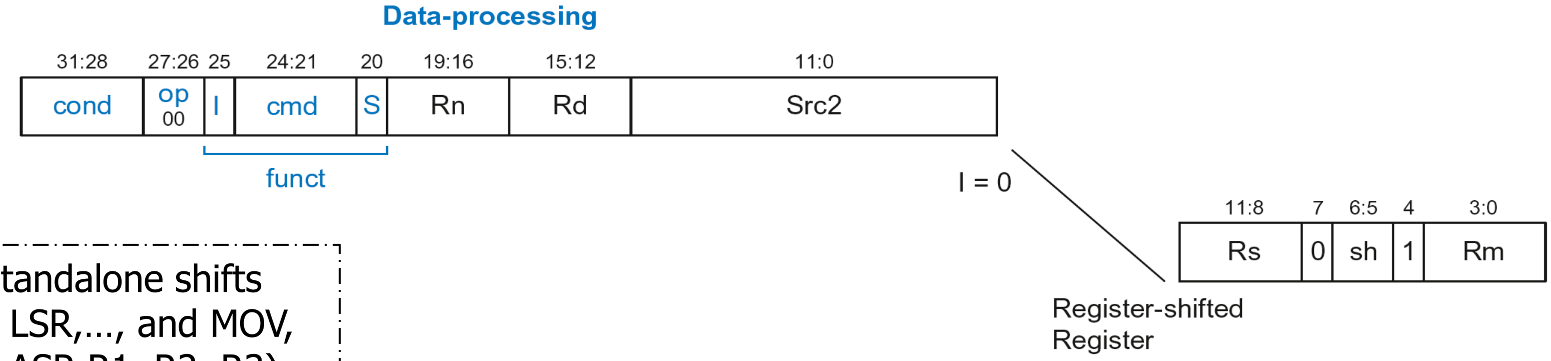
- ORR R8, R1, R3, LSR #2
  - Operation: R8 = R1 OR (R3 >> 2)
  - cond = "1110" (14) for unconditional execution
  - op = "00" (0) for data-processing instructions
  - cmd = "1100" (12) for ORR
  - *Src2* is a register so I=0
  - *Rd* = 8, *Rn* = 1, *Rm* = 3
  - *shamt5* = 2, *sh* = "01" (LSR)



DP with Register-shifted Register *Src2* : e.g.

<OP Rd, Rn, Rm, ST Rs>

- EOR R8, R9, R10, ROR R12
  - Operation: R8 = R9 XOR (R10 ROR R11)
  - cond = “1110” (14) for unconditional execution
  - op = “00” (0) for data-processing instructions
  - cmd = “0001” (1) for EOR
  - *Src2* is a register so I=0
  - *Rd* = 8, *Rn* = 9, *Rm* = 10, *Rs* = 12
  - *sh* = “11” (ROR)



Data-processing

31:28	27:26	25	24:21	20	19:16	15:12	11:0
cond	op 00	I	cmd	S	Rn	Rd	Src2

funct

I = 0

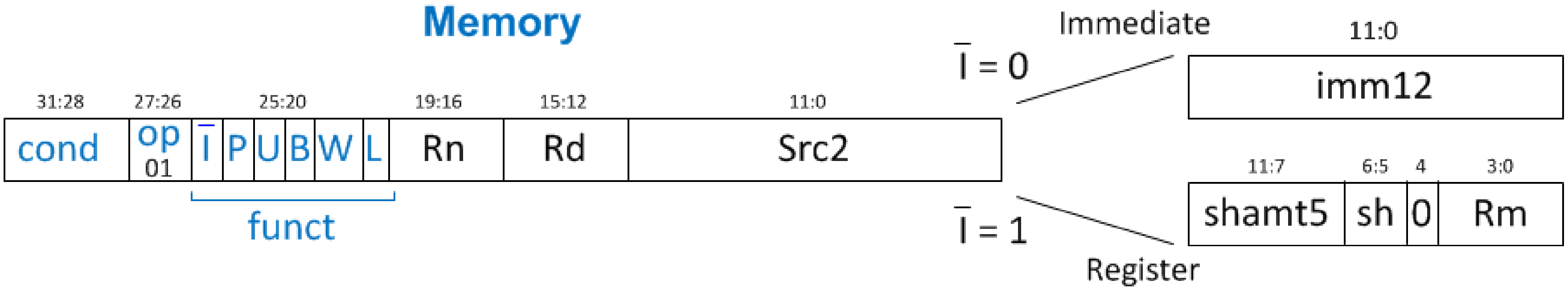
11:8	7	6:5	4	3:0
Rs	0	sh	1	Rm

Register-shifted Register

For standalone shifts ASR, LSR,…, and MOV, (ex : ASR R1, R2, R3) cmd=“1101” and *Rn*=0

Memory Instruction Format

- Encodes LDR, STR, LDRB, STRB
  - op : “01”
  - funct : 6 control bits
  - *Rn* : base register
  - *Rd* : destination (load), source (store)
  - *Src2* : offset
    - immediate <OP Rd, [Rn, #Imm12]>
    - register (optionally shifted) <OP Rd, [Rn, Rm, ST #shamt5]>



Memory

31:28	27:26	25:20						19:16	15:12	11:0
cond	op 01	$\overline{I}$	P	U	B	W	L	Rn	Rd	Src2

funct

$\overline{I} = 0$ Immediate

11:0
imm12

$\overline{I} = 1$ Register

11:7	6:5	4	3:0
shamt5	sh	0	Rm

Memory Instruction Format

- funct
  - $\overline{I}$: Immediate bar
  - P : Preindex
  - U : Add
  - B : Byte
  - W : Writeback
  - L : Load

P	*W*	**Indexing Mode**
0	1	Not supported
0	0	Postindex
1	0	Offset
1	1	Preindex

L	*B*	**Instruction**
0	0	`STR`
0	1	`STRB`
1	0	`LDR`
1	1	`LDRB`

```
LDR R8, [R1], #8
•   R8 = mem[R1]
•   R1 = R1 + 8
```

```
LDR R1, [R2, #4]
•  R1 = mem[R2+4]
```

```
LDR R3, [R5, #16]!
•  R3 = mem[R5+16]
•  R5 = R5 + 16
```

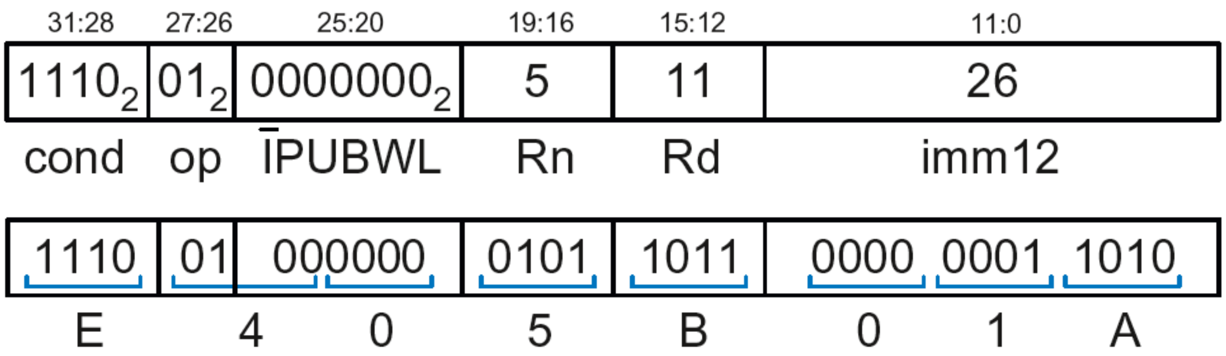
Value	$\overline{I}$	***U***
0	**Immediate** offset in *Src2*	**Subtract** offset from base
1	**Register** offset in *Src2*	**Add** offset to base

Memory Instr. with Immediate *Src2* : e.g.

<OP Rd, [Rn, #Imm12]>

- STR R11, [R5], #-26
  - Operation: mem[R5] <= R11; R5 = R5 - 26
  - cond = "1110" (14) for unconditional execution
  - op = "01" (1) for memory instruction
  - funct = "0000000" (0)
  - $\bar{I}$ = 0 (immediate offset), P = 0 (postindex), U = 0 (subtract), B = 0 (store word), W = 0 (postindex), L = 0 (store)
  - *Rd* = 11, *Rn* = 5, *Imm12* = 26

Field Values



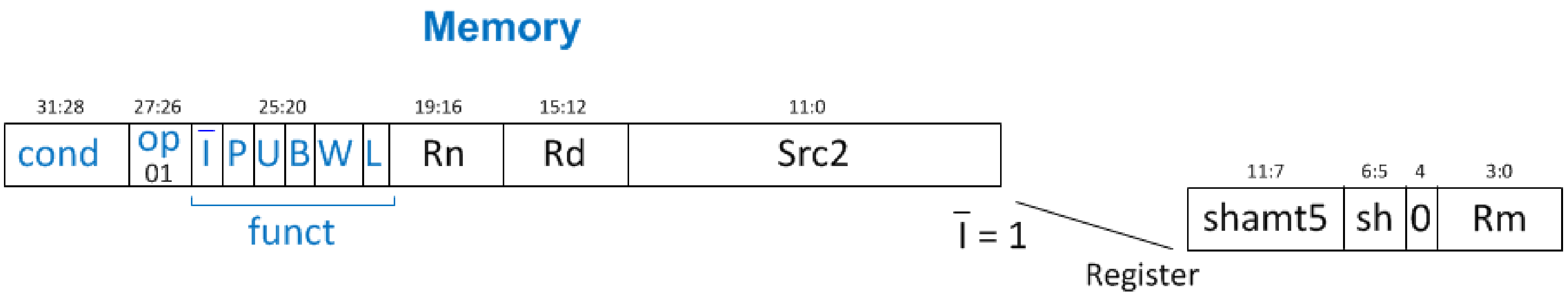
31:28	27:26	25:20	19:16	15:12	11:0
1110_2	01_2	0000000_2	5	11	26
cond	op	$\bar{I}$PUBWL	Rn	Rd	imm12

1110	01	000000	0101	1011	0000 0001 1010
E	4	0	5	B	0 1 A

Memory Instr. with Scaled Reg. *Src2* : e.g.

<OP Rd, [Rn, Rm, ST #shamt5]>

- STR R9, [R1, R3, LSL #2]
  - Operation: mem[R1 + (R3 << 2)] <= R9
  - cond = "1110" (14) for unconditional execution
  - op = "01" (1) for memory instruction
  - funct = "111000" (56)
  - $\overline{I}$ = 1 (register offset), P = 1 (offset indexing), U = 1 (add), B = 0 (store word), W = 0 (offset indexing), L = 0 (store)
  - *Rd* = 9, *Rn* = 1, *Rm* = 3, *shamt* = 2, *sh* = "00" (LSL)



Memory

31:28 cond | 27:26 op 01 | 25:20 $\overline{I}$ P U B W L (funct) | 19:16 Rn | 15:12 Rd | 11:0 Src2

$\overline{I}$ = 1 Register

11:7 shamt5 | 6:5 sh | 4 0 | 3:0 Rm

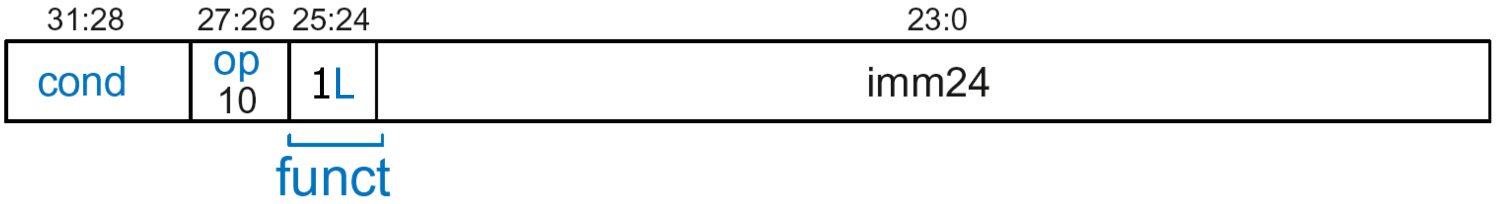
Branch Instruction Format

- Encodes B and BL

<OP Imm24>

  - op = “10”
  - funct = “1L” : L = 1 for BL, L = 0 for B
  - *Imm24* : 24-bit immediate encoding Branch Target Address (BTA)
    - BTA = Next PC when branch taken
    - Imm24 = # of words BTA is away from current PC+8
    - Note : (# of words) = (# of bytes / 4) = (# of bytes >> 2)

Branch



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

funct

Branch Instruction : e.g.

```
0x8040 TEST: LDRB R5, [R0, R3] ←BTA
0x8044       STRB R5, [R1, R3]
0x8048       ADD  R3, R3, #1
0x804C       MOV  PC, LR
0x8050       BL   TEST          ← PC
0x8054       LDR  R3, [R1], #4
0x8058       SUB  R4, R3, #9    ←PC+8
```

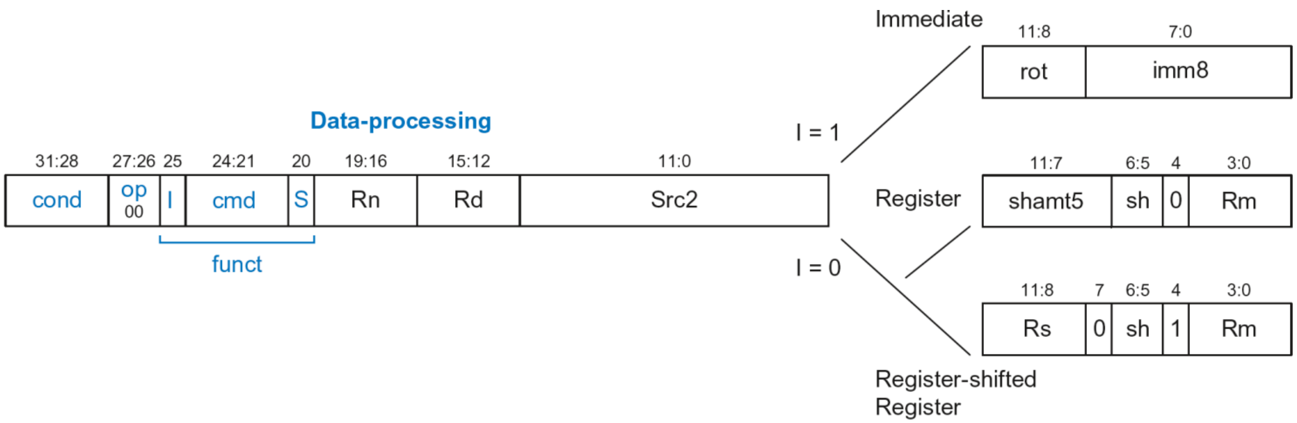
- PC = 0x8050
- PC + 8 = 0x8058
- BTA = 0x8040
- `TEST` label is 6 instructions before PC+8, so *Imm24* = -6
  - Offset = 0x8040-0x8058 = -0x18 bytes = -0x6 words

Field Values

31:28	27:26	25:24	23:0
1110_2	10_2	11_2	-6
cond	op	funct	imm24

1110	10	11	1111 1111 1111 1111 1111 1010	0xEBFFFFFA

Review: Instruction Formats



Data-processing

31:28	27:26	25	24:21	20	19:16	15:12	11:0
cond	op 00	I	cmd	S	Rn	Rd	Src2

funct

I = 1

Immediate

11:8	7:0
rot	imm8

<OP Rd, Rn, #Imm8_rot>

I = 0

Register

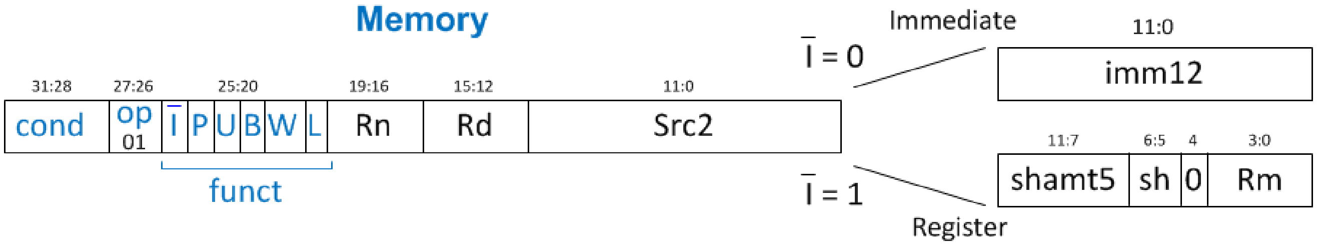
11:7	6:5	4	3:0
shamt5	sh	0	Rm

<OP Rd, Rn, Rm, ST #shamt5>

Register-shifted Register

11:8	7	6:5	4	3:0
Rs	0	sh	1	Rm

<OP Rd, Rn, Rm, ST Rs>



Memory

31:28	27:26	25:20						19:16	15:12	11:0
cond	op 01	$\bar{I}$	P	U	B	W	L	Rn	Rd	Src2

funct

$\bar{I}$ = 0

Immediate

11:0
imm12

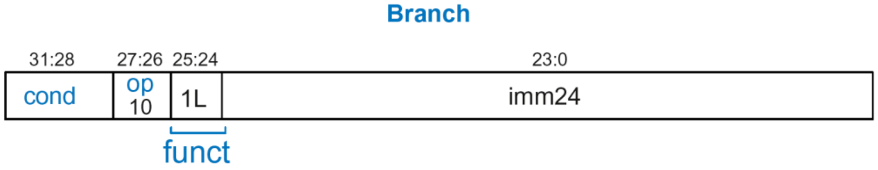
<OP Rd, [Rn, #Imm12]>

$\bar{I}$ = 1

Register

11:7	6:5	4	3:0
shamt5	sh	0	Rm

<OP Rd, [Rn, Rm, ST #shamt5]>



Branch

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

funct

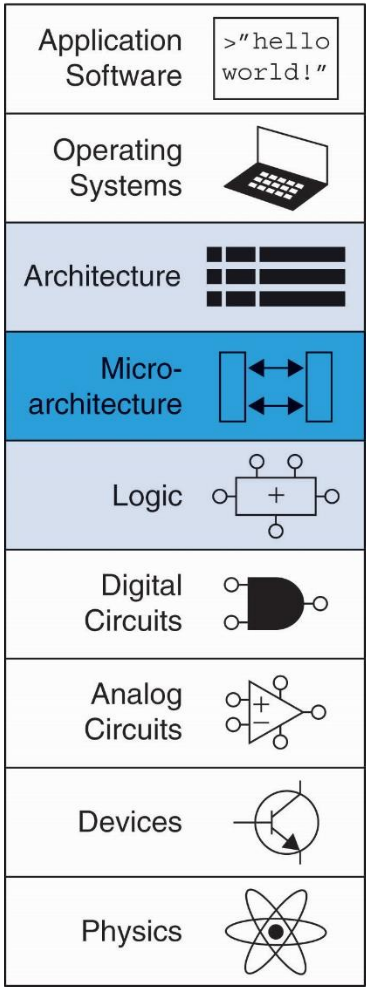
<OP Imm24>

Take a printout of this page and keep it for quick reference for the rest of this chapter

Microarchitecture

- Processor
  - Datapath: functional blocks
  - Control: control signals

- Basic styles
  - **Single-cycle**: Each instruction executes in a single cycle
  - **Multicycle**: Each instruction is broken up into series of shorter steps
  - **Pipelined**: Each instruction broken up into series of steps & multiple instructions execute at once



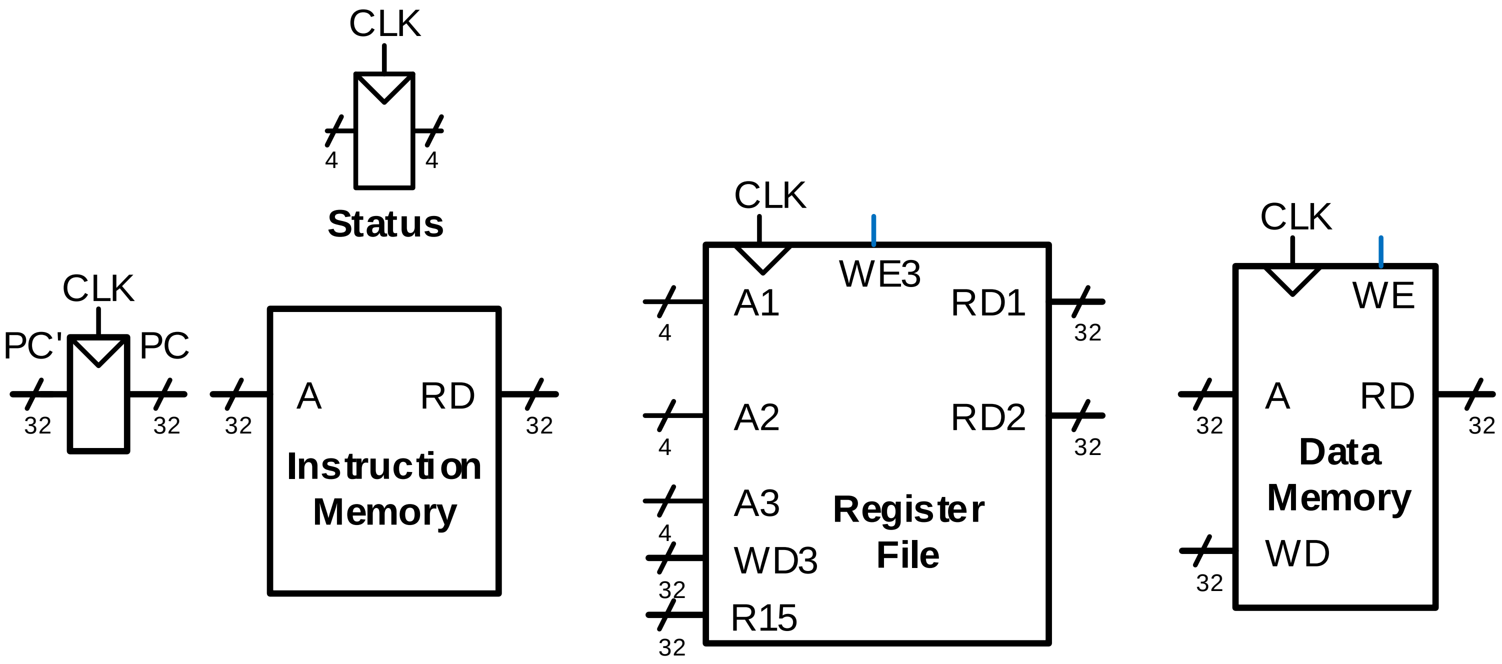
ARM Processor : Getting Started

- We will get started with a subset of ARM instructions

- Data-processing instructions
  - ADD, SUB, AND, ORR
  - with register and immediate *Src2*, but no shifts
- Memory instructions
  - LDR, STR
  - with positive immediate offset
- Branch instructions
  - B

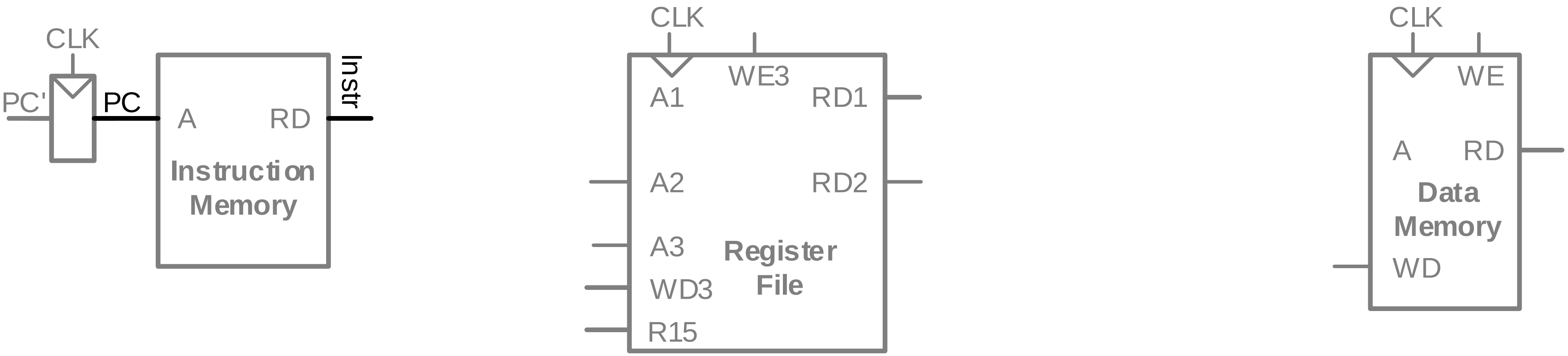
Architectural State Elements

- Determines everything about a processor
- Architectural state
  - 16 registers (including PC) –> Register File (RF)
  - Status register
- Memory



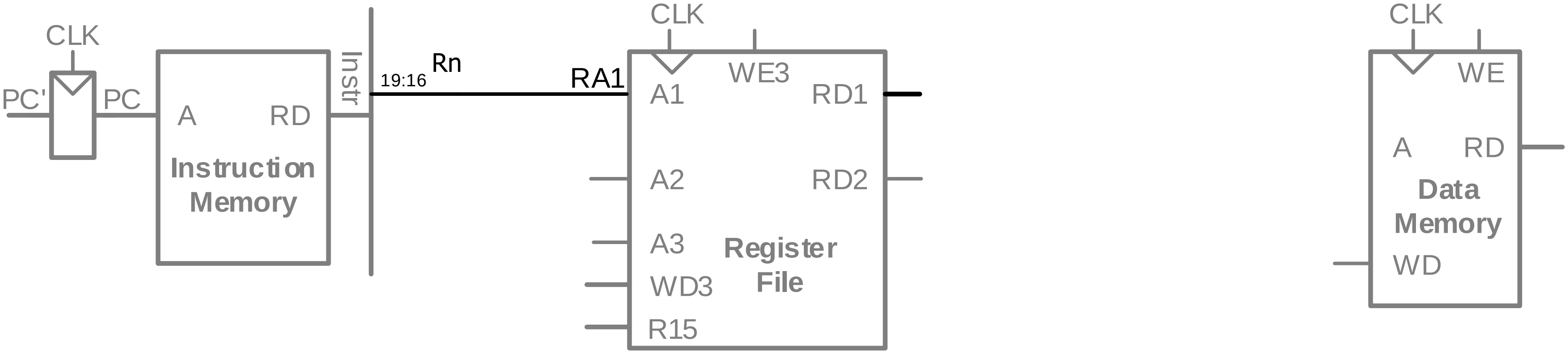
Single-Cycle Datapath: LDR Fetch

- STEP 1: Fetch instruction



Single-Cycle Datapath: LDR Reg Read

- STEP 2: Read source operands from RF



Memory

31:28	27:26	25:20						19:16	15:12	11:0
cond	op 01	$\bar{I}$	P	U	B	W	L	Rn	Rd	Src2
		funct								

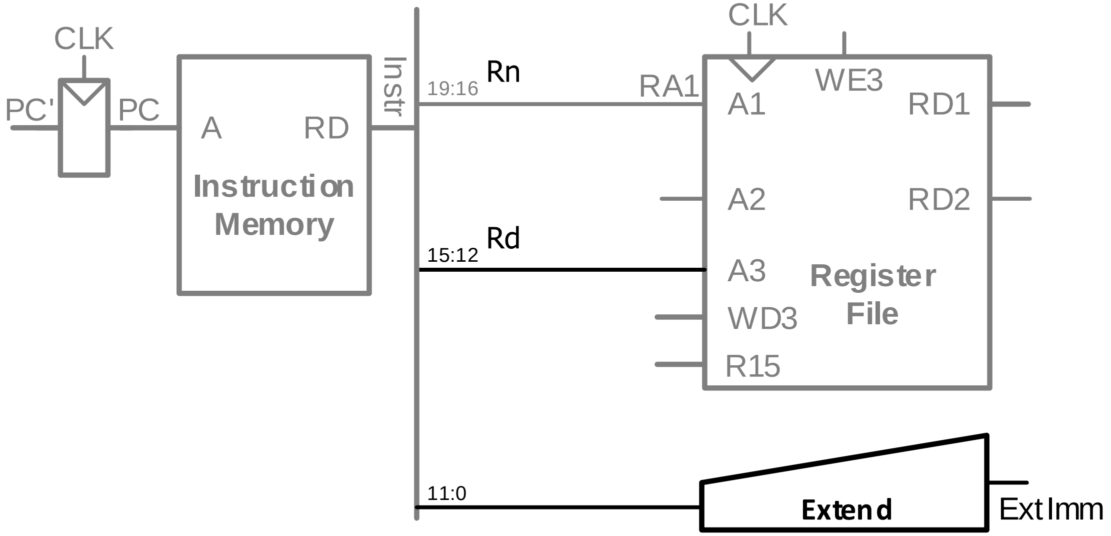
$\bar{I}$ = 0 Immediate

11:0
imm12

<LDR Rd, [Rn, #Imm12]>

Single-Cycle Datapath: LDR Immed.

- STEP 3: Extend the immediate



Memory

31:28	27:26	25:20						19:16	15:12	11:0
cond	op 01	$\overline{I}$	P	U	B	W	L	Rn	Rd	Src2
		funct								

$\overline{I}$ = 0

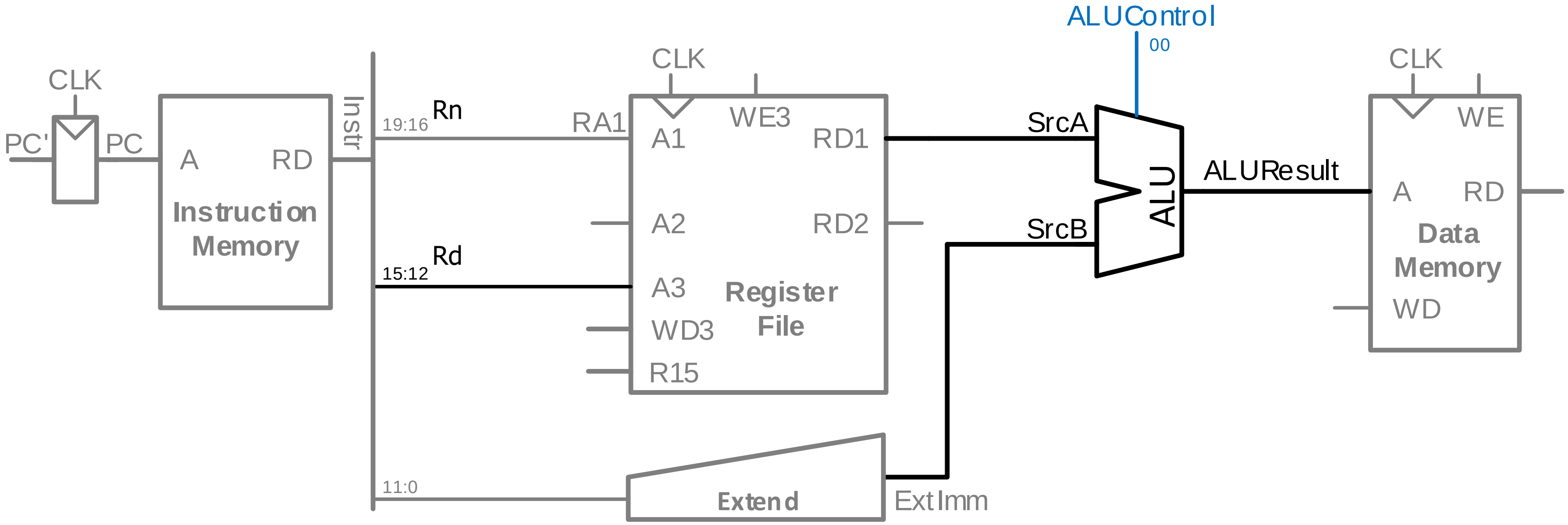
Immediate

11:0
imm12

<LDR Rd, [Rn, #Imm12]>

Single-Cycle Datapath: LDR Address

- STEP 4: Compute the memory address



Memory

31:28	27:26	25:20						19:16	15:12	11:0
cond	op 01	Ī	P	U	B	W	L	Rn	Rd	Src2

funct

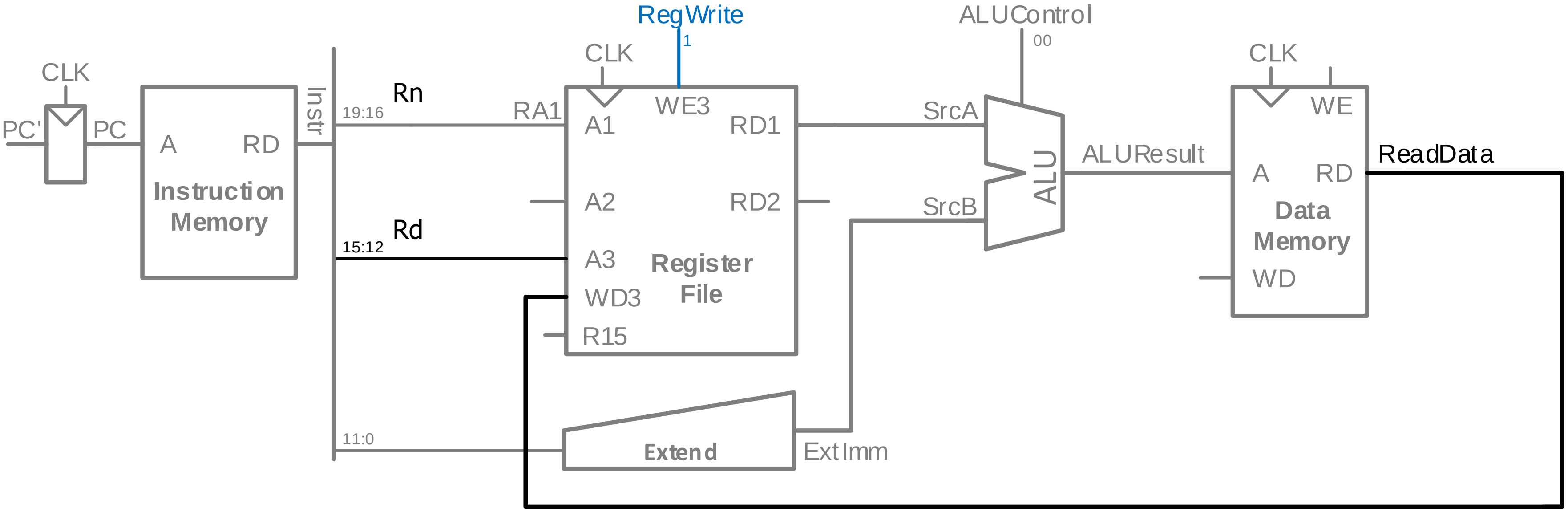
Ī = 0 Immediate

11:0
imm12

<LDR Rd, [Rn, #Imm12]>

Single-Cycle Datapath: LDR Mem Read

- STEP 5: Read data from memory and write it back to register file



Memory

31:28	27:26	25:20						19:16	15:12	11:0
cond	op 01	Ī	P	U	B	W	L	Rn	Rd	Src2
		funct								

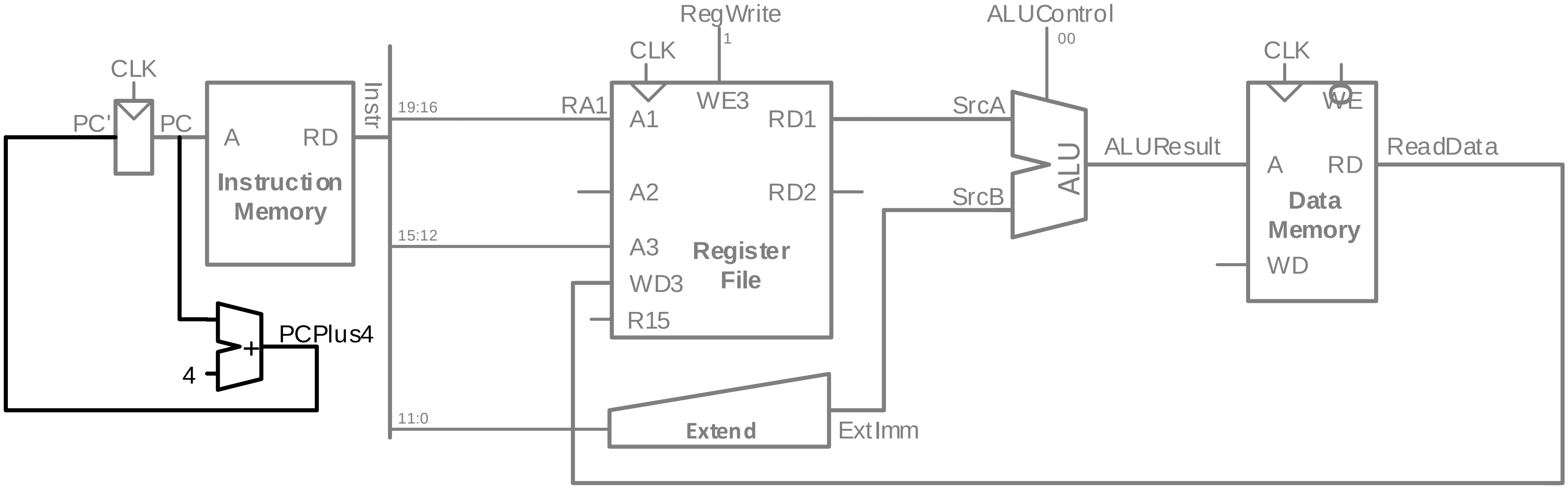
Ī = 0

Immediate	11:0
	imm12

<LDR Rd, [Rn, #Imm12]>

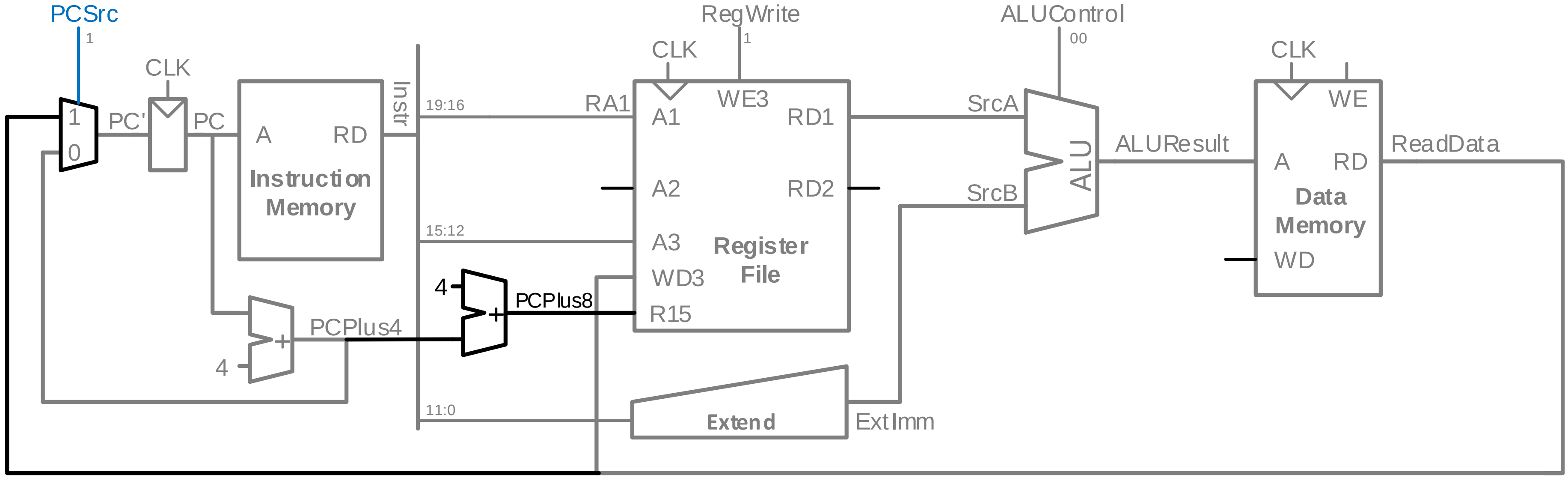
Single-Cycle Datapath: PC Increment

- STEP 6: Determine address of next instruction



Single-Cycle Datapath: PC Source/Dest

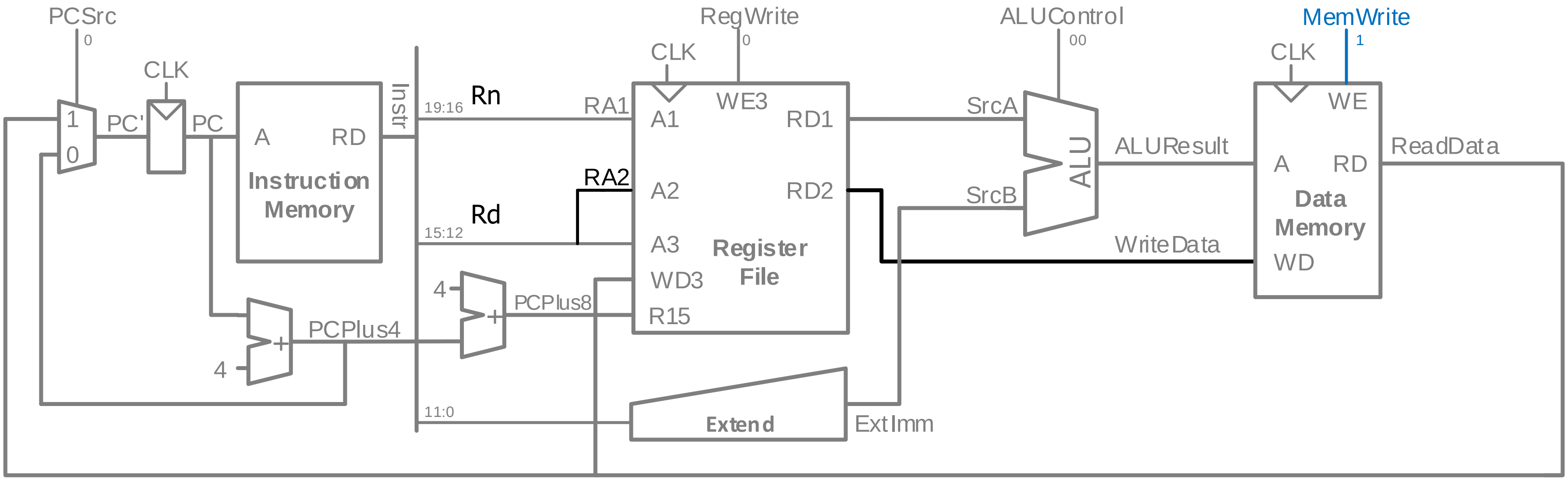
- PC can be source/destination of instruction
  - Source: R15 (PC+8) available in Register File
  - Destination: Be able to write result to PC



R15 is not "stored" in the register file. If A1 or A2 is 15, RD1 or RD2 gets the value of PC after being passed through two adders (i.e., PC+8)

Single-Cycle Datapath: STR

- Write data in RD to memory
  - Note that *Rd* is a source operand for STR



Memory

31:28	27:26	25:20						19:16	15:12	11:0
cond	op 01	Ī	P	U	B	W	L	Rn	Rd	Src2
		funct								

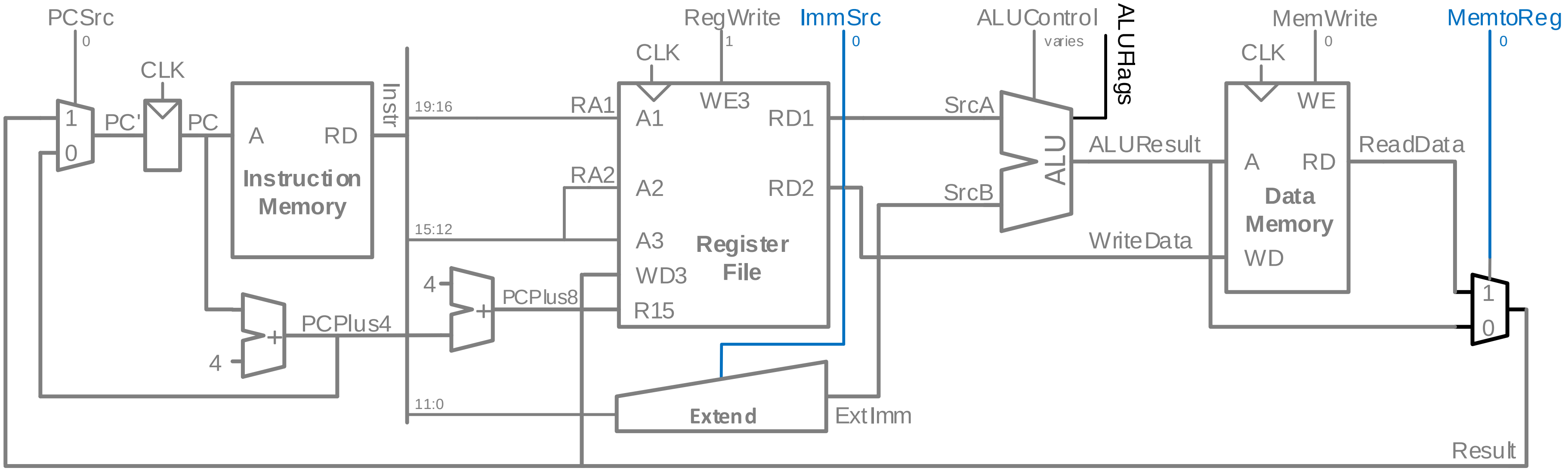
Ī = 0

Immediate
11:0
imm12

<STR Rd, [Rn, #Imm12]>

Single-Cycle Datapath: Data Processing

- With immediate *Src2*
  - Read from *Rn* and *Imm8* (ImmSrc chooses the zero-extended *Imm8* instead of *Imm12*)
  - Write ALUResult to register file (*Rd*)



Immediate

11:8	7:0
rot	imm8

I = 1

<OP Rd, Rn, #Imm8~~_rot~~>

^

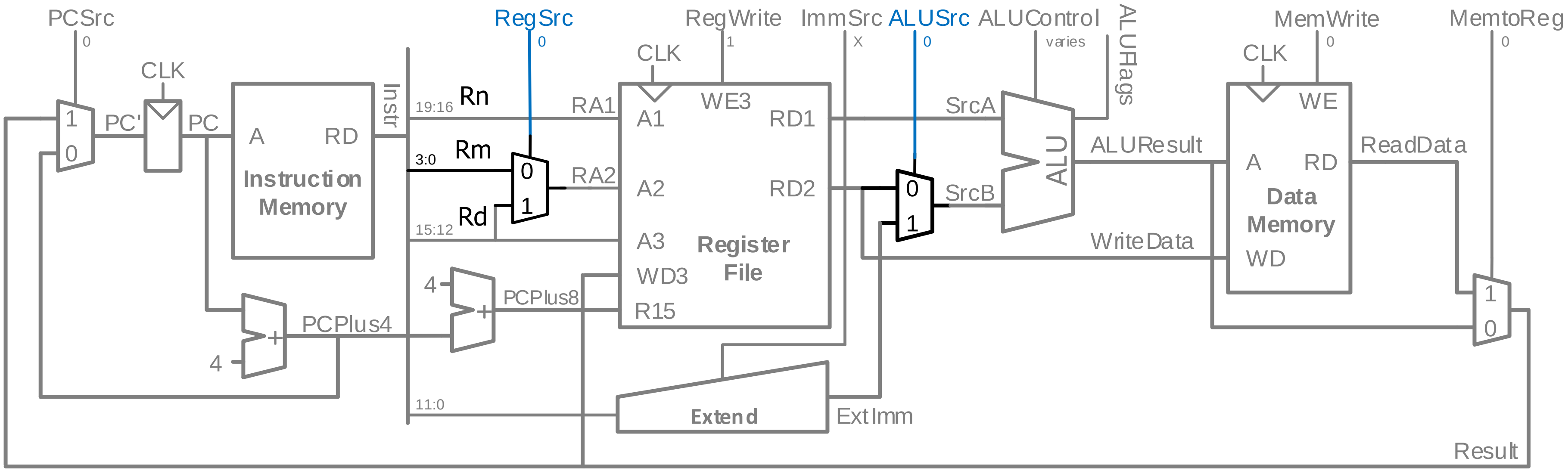
We're only considering DP with no shifts for Src2 reg.

Data-processing

31:28	27:26	25	24:21	20	19:16	15:12	11:0
cond	op 00	I	cmd	S	Rn	Rd	Src2

Single-Cycle Datapath: Data Processing

- With register *Src2*
  - Read from *Rn* and *Rm* (instead of *Imm8*)



Data-processing

31:28	27:26	25	24:21	20	19:16	15:12	11:0
cond	op 00	I	cmd	S	Rn	Rd	Src2
		funct					

I = 0

Register

11:7	6:5	4	3:0
shamt5	sh	0	Rm

<OP Rd, Rn, Rm, ~~ST #shamt5~~>

We're only considering DP with no shifts for Src2 reg.

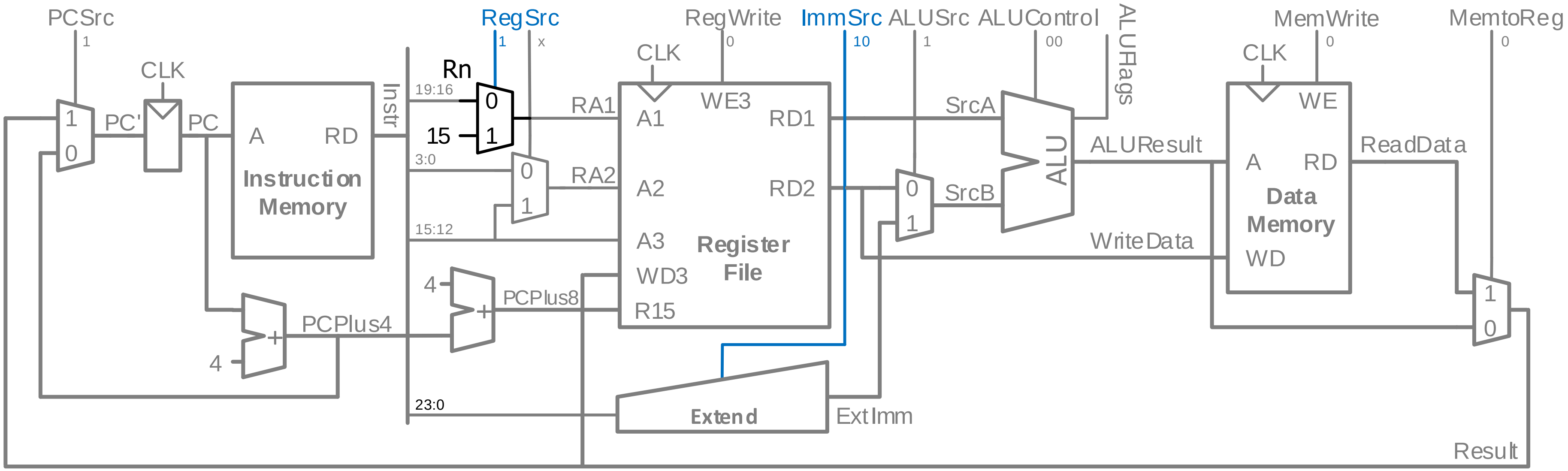
Register-shifted Register

11:8	7	6:5	4	3:0
Rs	0	sh	1	Rm

<OP Rd, Rn, Rm, ~~ST Rs~~>

Single-Cycle Datapath: Branch

- Calculate branch target address
  - BTA = (ExtImm) + (PC + 8)
  - ExtImm = (sign-extended *Imm24*) << 2



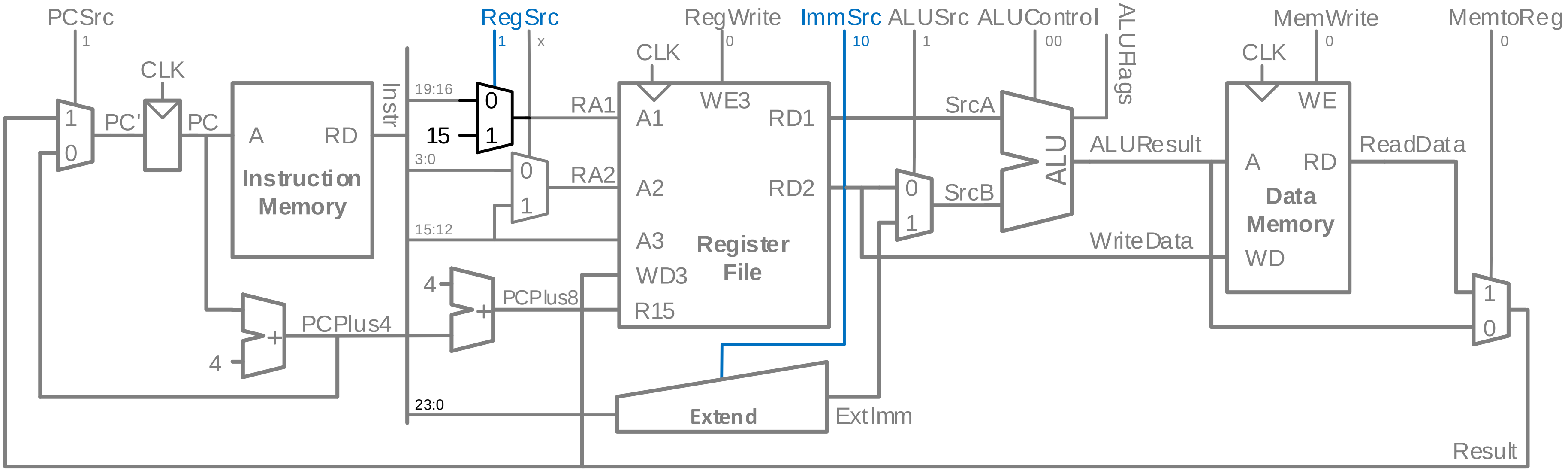
Branch

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

funct

<OP Imm24>

Single-Cycle Datapath: ExtImm

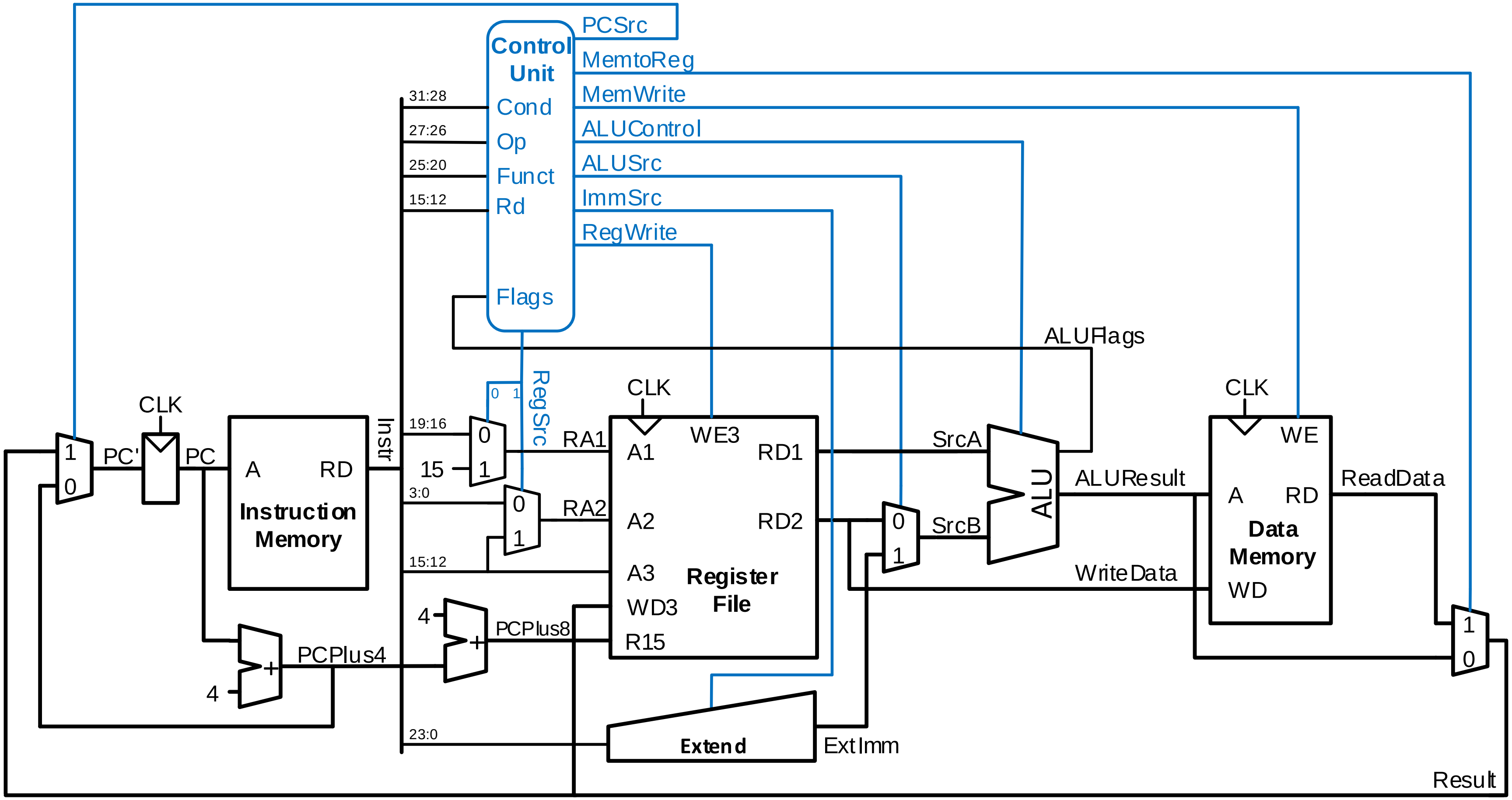


$\text{ImmSrc}_{1:0}$	ExtImm	Description
00	(24{'0'}, $\text{Instr}_{7:0}$)	Zero-extended *Imm8*
01	(20{'0'}, $\text{Instr}_{11:0}$)	Zero-extended *Imm12*
10	(6{Instr_{23}}, $\text{Instr}_{23:0}$, 2{'0'})	(Sign-extended *Imm24*)<<2

How would you extend LDR and STR to support negative offsets?. Do you need sign extension?

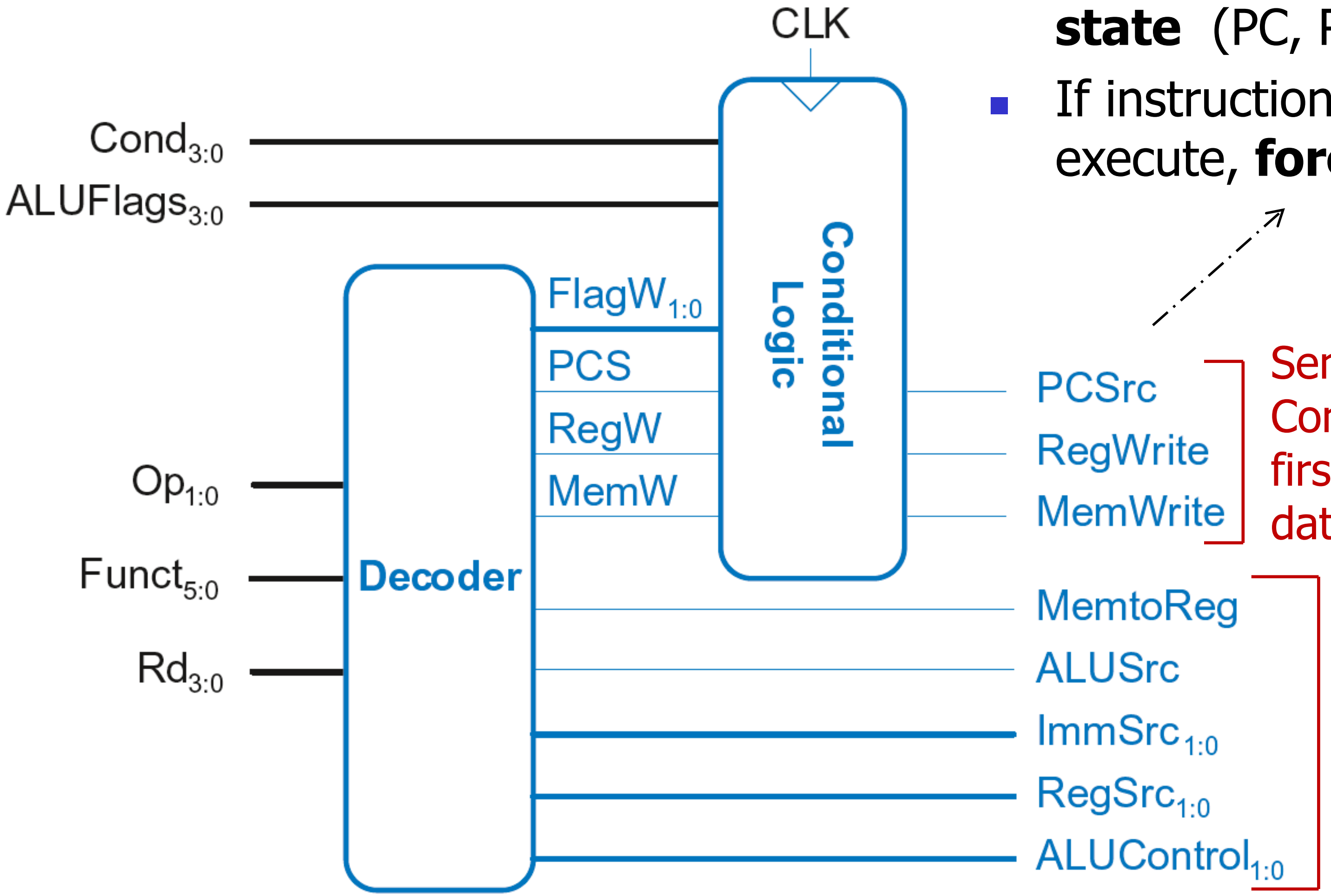
Q : Do you need a shifter to shift Imm24?
Ans : NO!. A shifter is required only if the shift amount is variable.
Shifting by a constant requires no logic gates -> can be accomplished by appropriate wiring!

Single-Cycle ARM Processor with Control



Single-Cycle Control

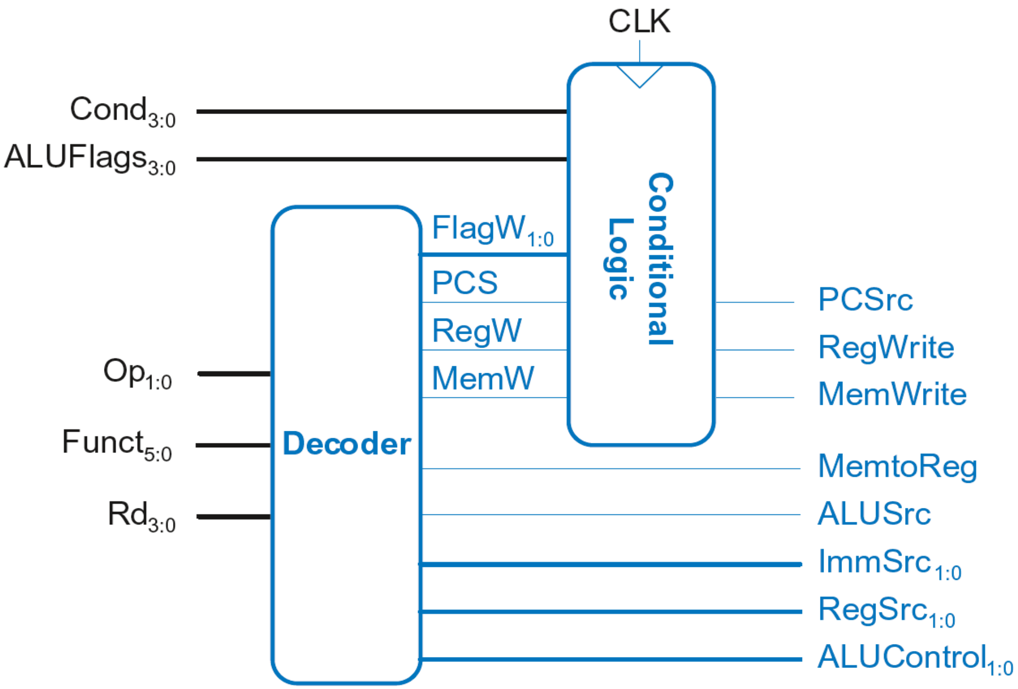
- These signals **change the state** (PC, RF, Memory)
- If instruction shouldn't execute, **forced to 0**



Sent through Conditional Logic first, then to datapath

Sent directly to datapath

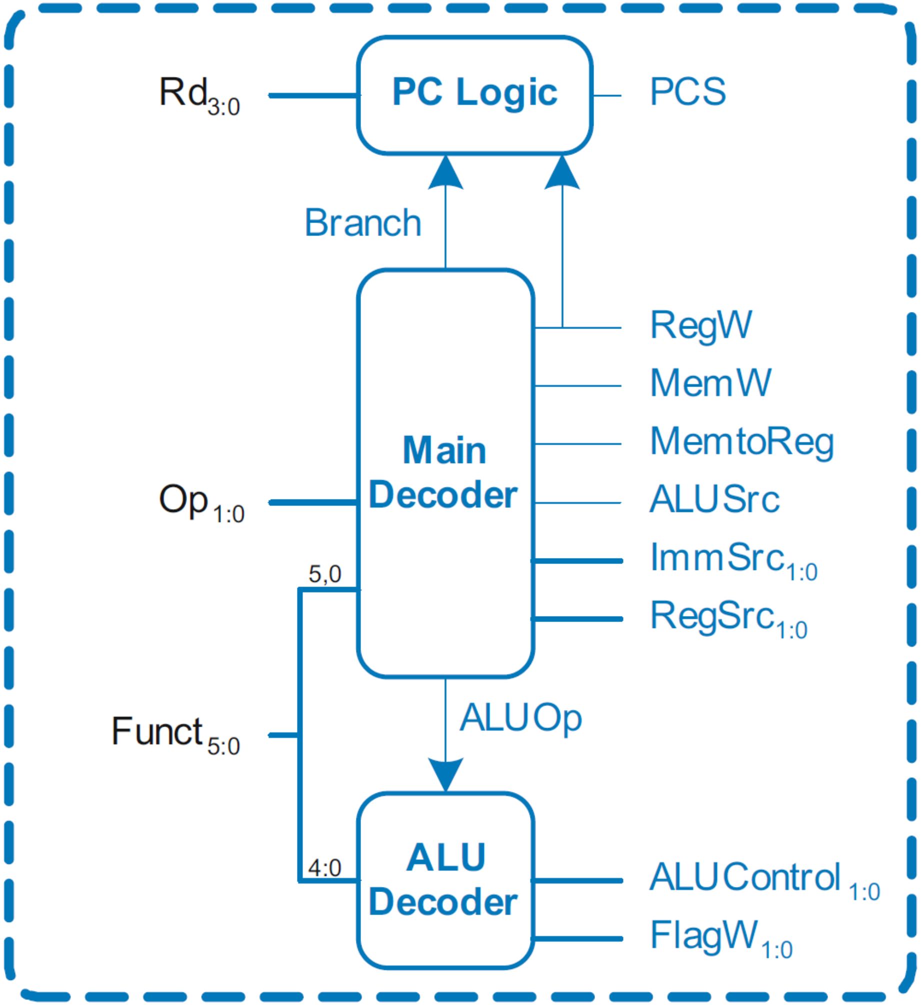
Single-Cycle Control



- $FlagW_{1:0}$: Flag Write signal, asserted when ALUFlags should be saved (i.e., on instruction with S=1)
- `ADD`, `SUB` update all flags (NZCV)
- `AND`, `ORR` only update NZ flags
- So, two bits needed:
  - $FlagW_1$ = 1: NZ saved ($ALUFlags_{3:2}$ saved)
  - $FlagW_0$ = 1: CV saved ($ALUFlags_{1:0}$ saved)

Single-Cycle Control: Decoder

- Submodules
    - Main Decoder
    - ALU Decoder
    - PC Logic



Control Unit: Main Decoder

Op	Funct_5	Funct_0	Type	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp
00	0	X	DP Reg	0	0	0	0	XX	1	00	1
00	1	X	DP Imm	0	0	0	1	00	1	X0	1
01	X	0	`STR`	0	X	1	1	01	0	10	0
01	X	1	`LDR`	0	1	0	1	01	1	X0	0
10	X	X	`B`	1	0	0	1	10	0	X1	0

To make full sense of this slide, read it together with slides 22, 38 and 41

Control Unit: ALU Decoder

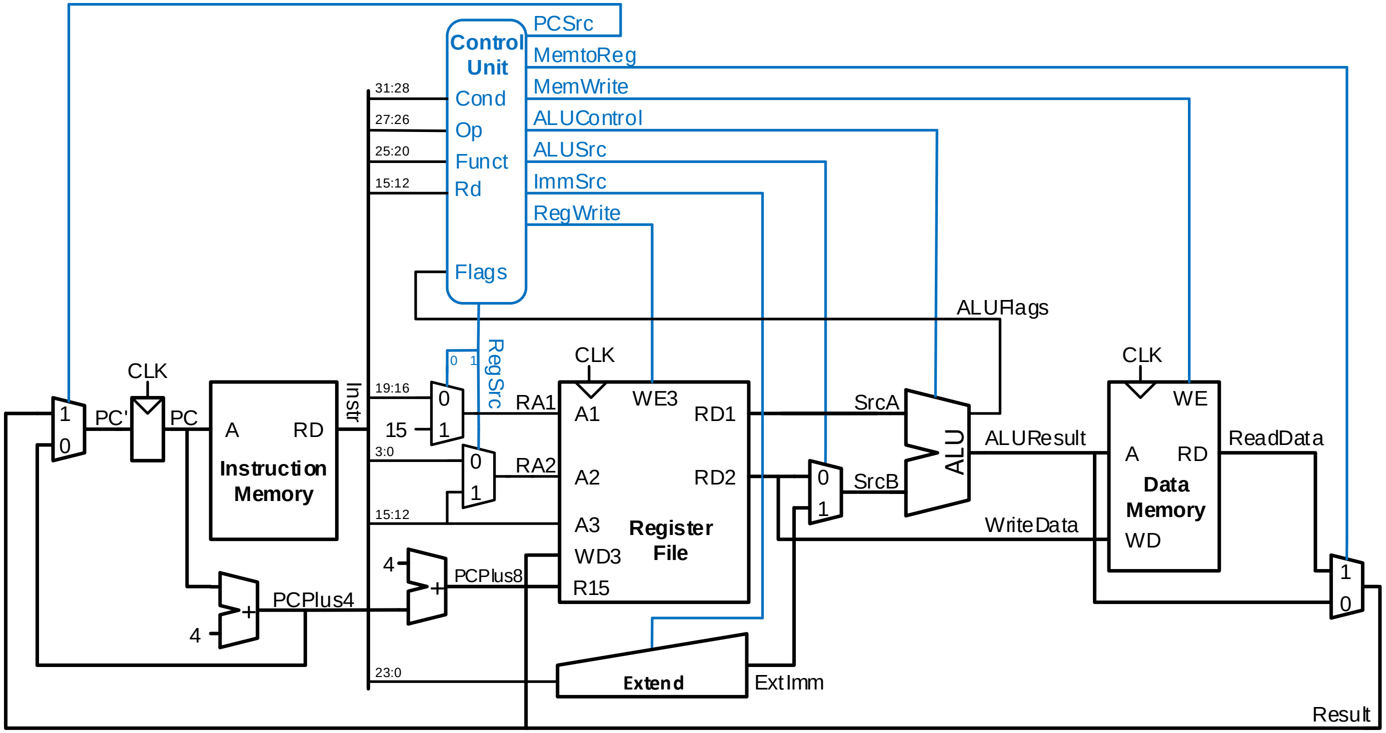
ALUOp	$Funct_{4:1}$ (*cmd*)	$Funct_0$ (*S*)	Type	$ALUControl_{1:0}$	$FlagW_{1:0}$
0	X	X	Not DP	00	00
1	0100	0	ADD	00	00
		1			11
	0010	0	SUB	01	00
		1			11
	0000	0	AND	10	00
		1			10
	1100	0	ORR	11	00
		1			10

- $FlagW_1$ = 1: NZ ($Flags_{3:2}$) should be saved
- $FlagW_0$ = 1: CV ($Flags_{1:0}$) should be saved*

* In reality, AND & ORR affects C as well. Omitted here for simplicity as Src2 does not supports shifts

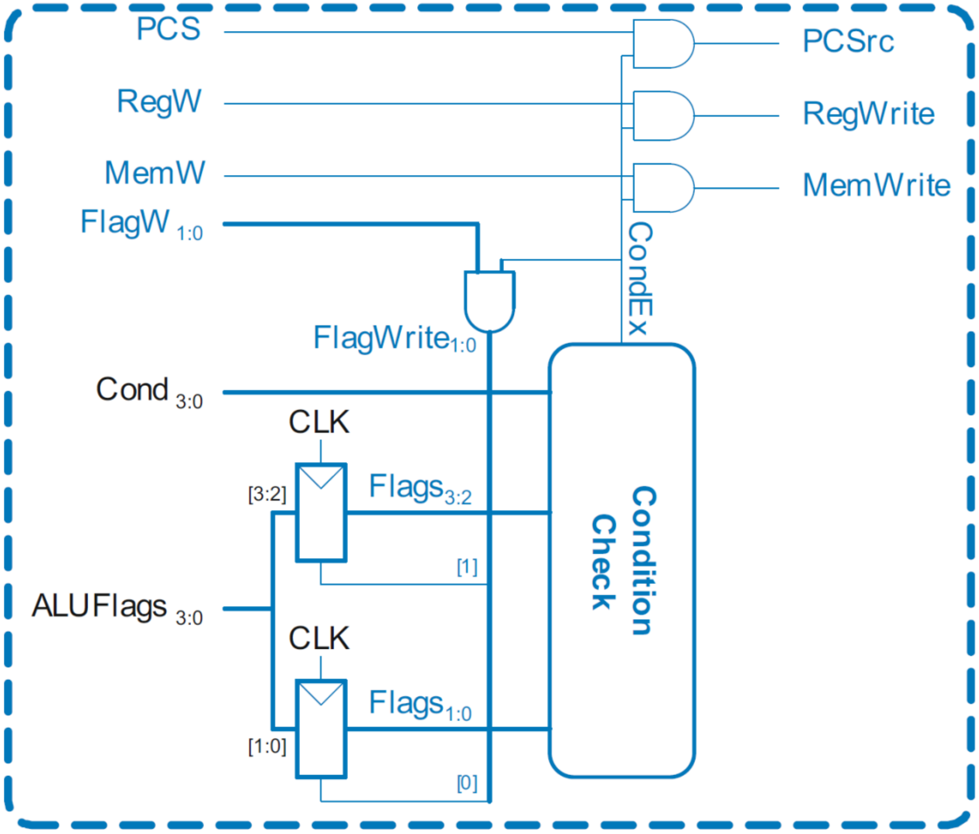
Single-Cycle Control: PC Logic

- PCS = 1 if PC is written by an instruction or branch (B):
  PCS = ((*Rd* == 15) & *RegW*) | *Branch*



- If instruction is executed: PCSrc = PCS
  Else PCSrc = 0 (i.e., PC = PC + 4)

Single-Cycle Control: Conditional Logic

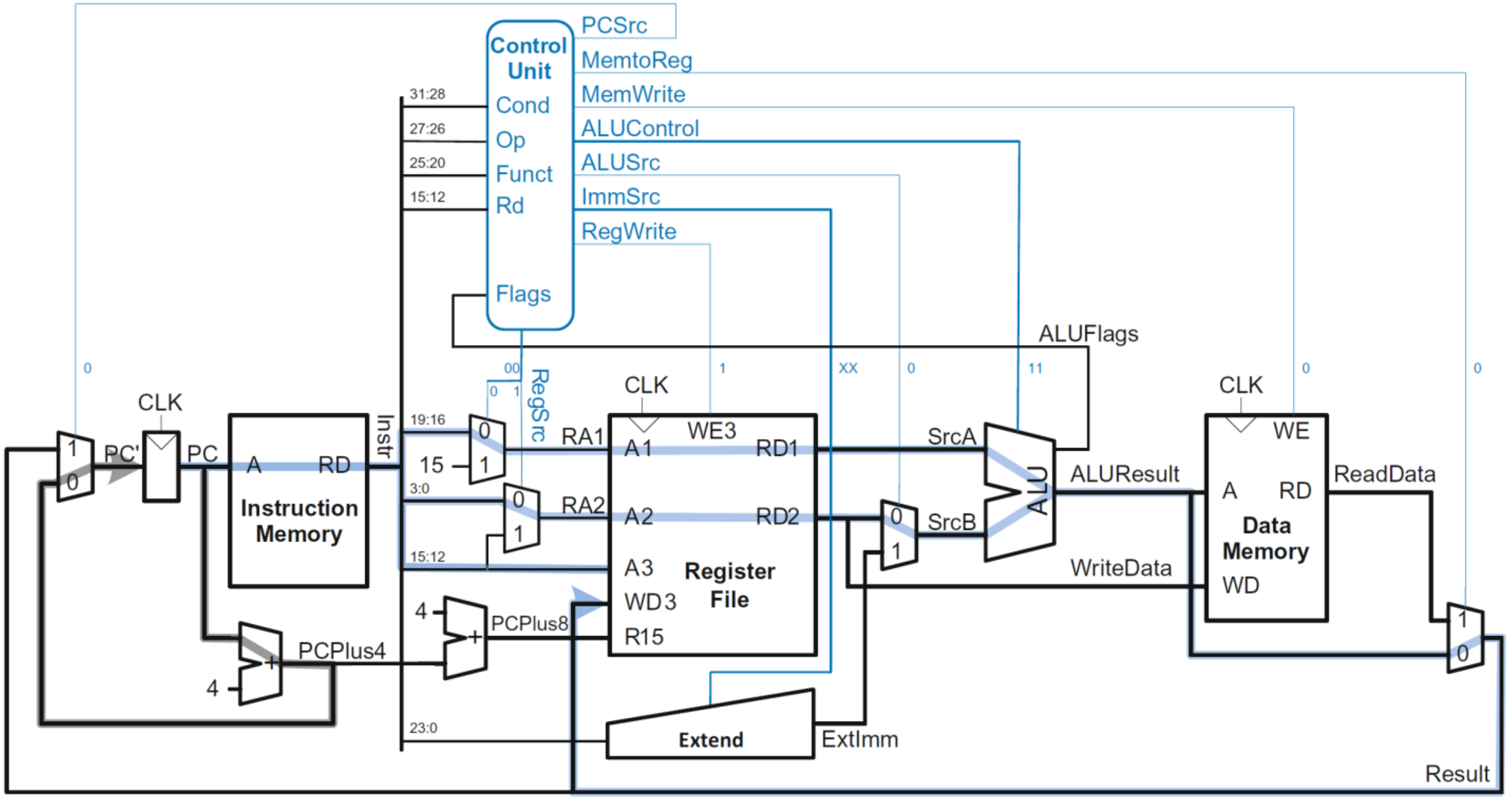


Recall:
ADD, SUB update all flags
AND, OR update NZ only.
So Flags status register has two write enables.
$\text{Flags}_{3:0}$ = {N,Z,C,V}

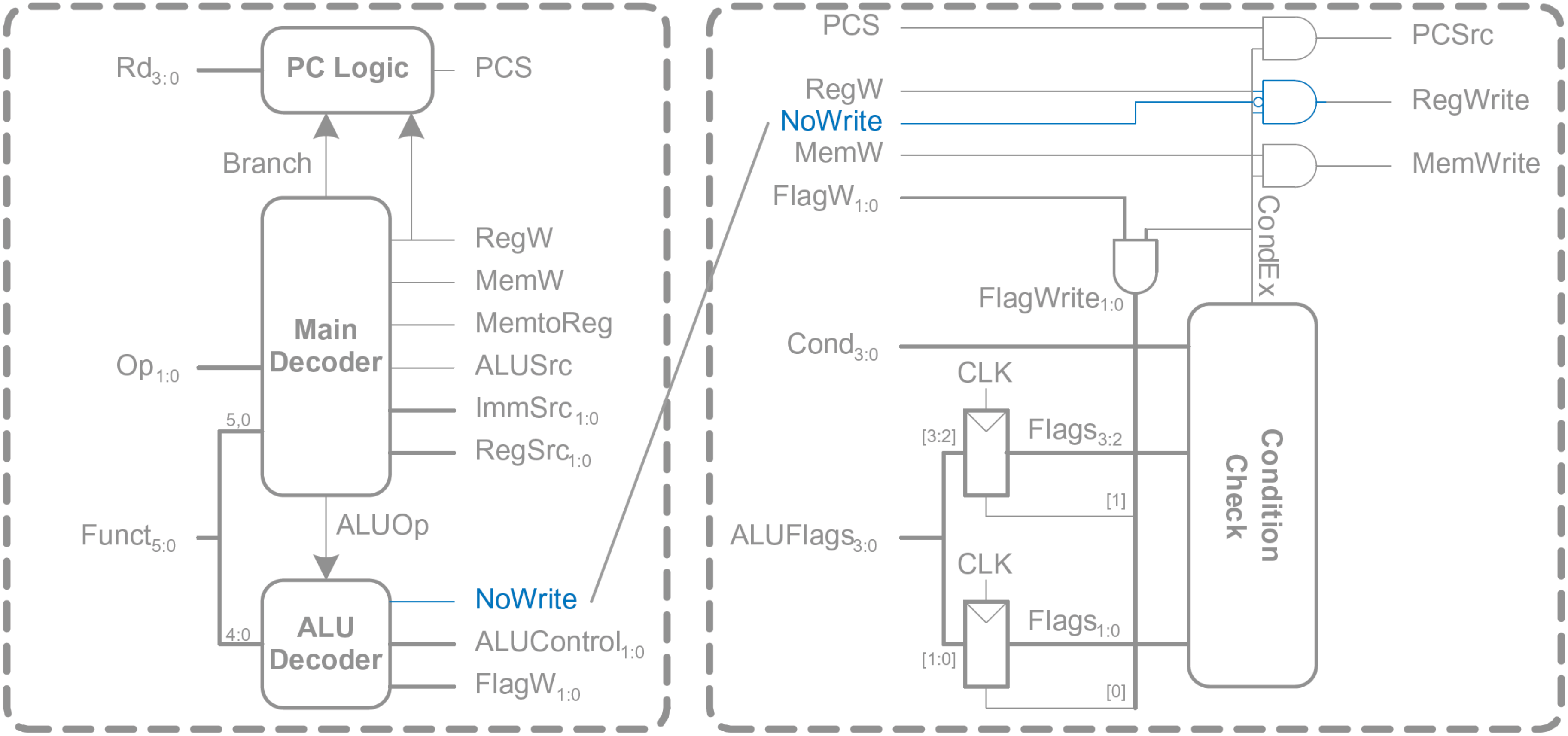
- Depending on condition mnemonic ($\text{Cond}_{3:0}$) and condition flags ($\text{Flags}_{3:0}$) the instruction is executed (CondEx = 1)

Example: ORR

Op	$Funct_5$	$Funct_0$	Type	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp
00	0	X	DP Reg	0	0	0	0	XX	1	00	1



Extended Functionality: CMP

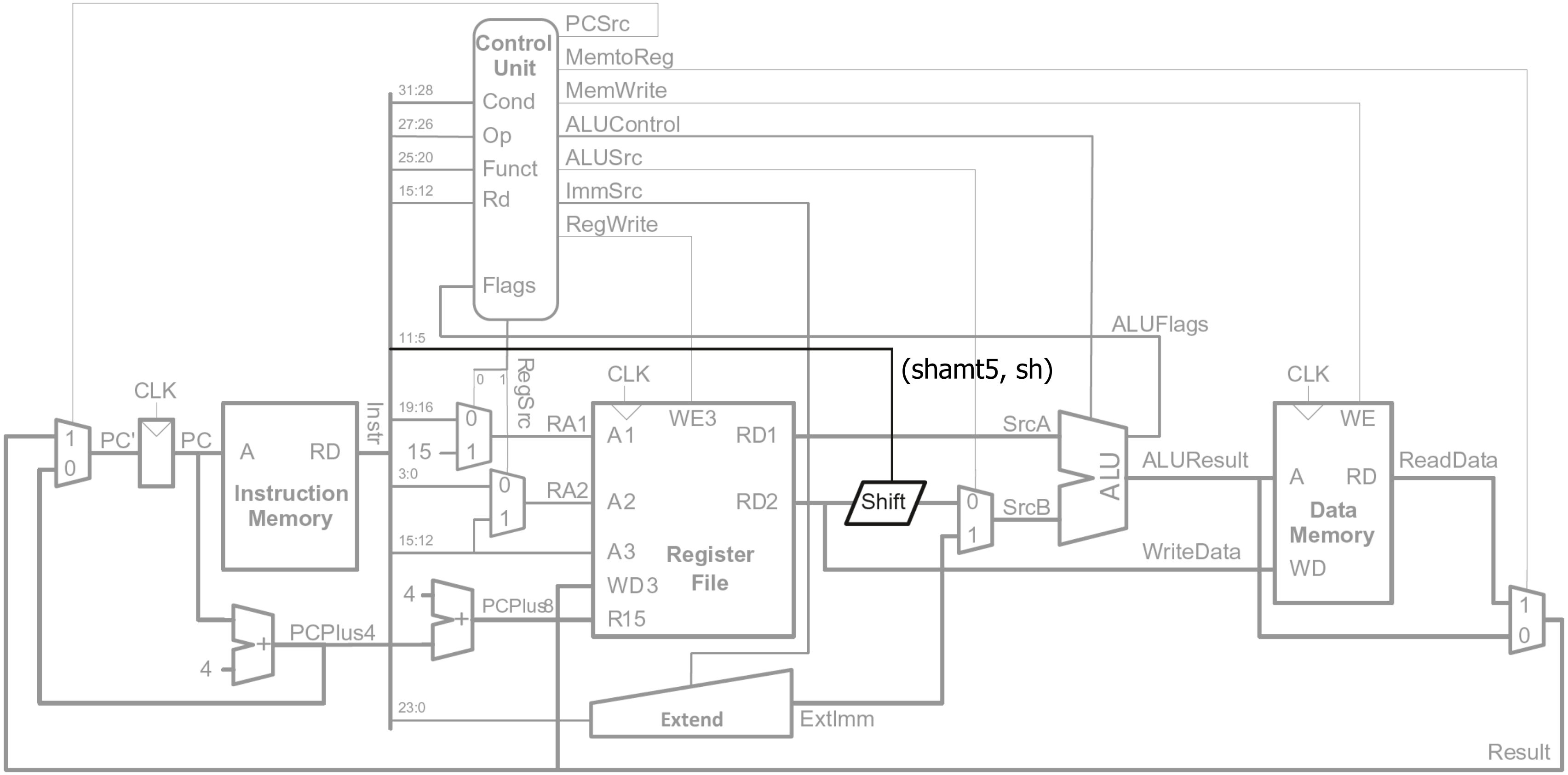


- Recall : CMP does subtraction, sets the flags and discards the result
- No change to datapath needed for implementation!

Extended Functionality: CMP

ALUOp	$Funct_{4:1}$ (*cmd*)	$Funct_0$ (*S*)	Type	$ALUControl_{1:0}$	$FlagW_{1:0}$	NoWrite
0	X	X	Not DP	00	00	0
1	0100	0	ADD	00	00	0
		1			11	0
	0010	0	SUB	01	00	0
		1			11	0
	0000	0	AND	10	00	0
		1			10	0
	1100	0	ORR	11	00	0
		1			10	0
	1010	**1**	**CMP**	**01**	**11**	**1**

Extended Functionality: Shifted Register

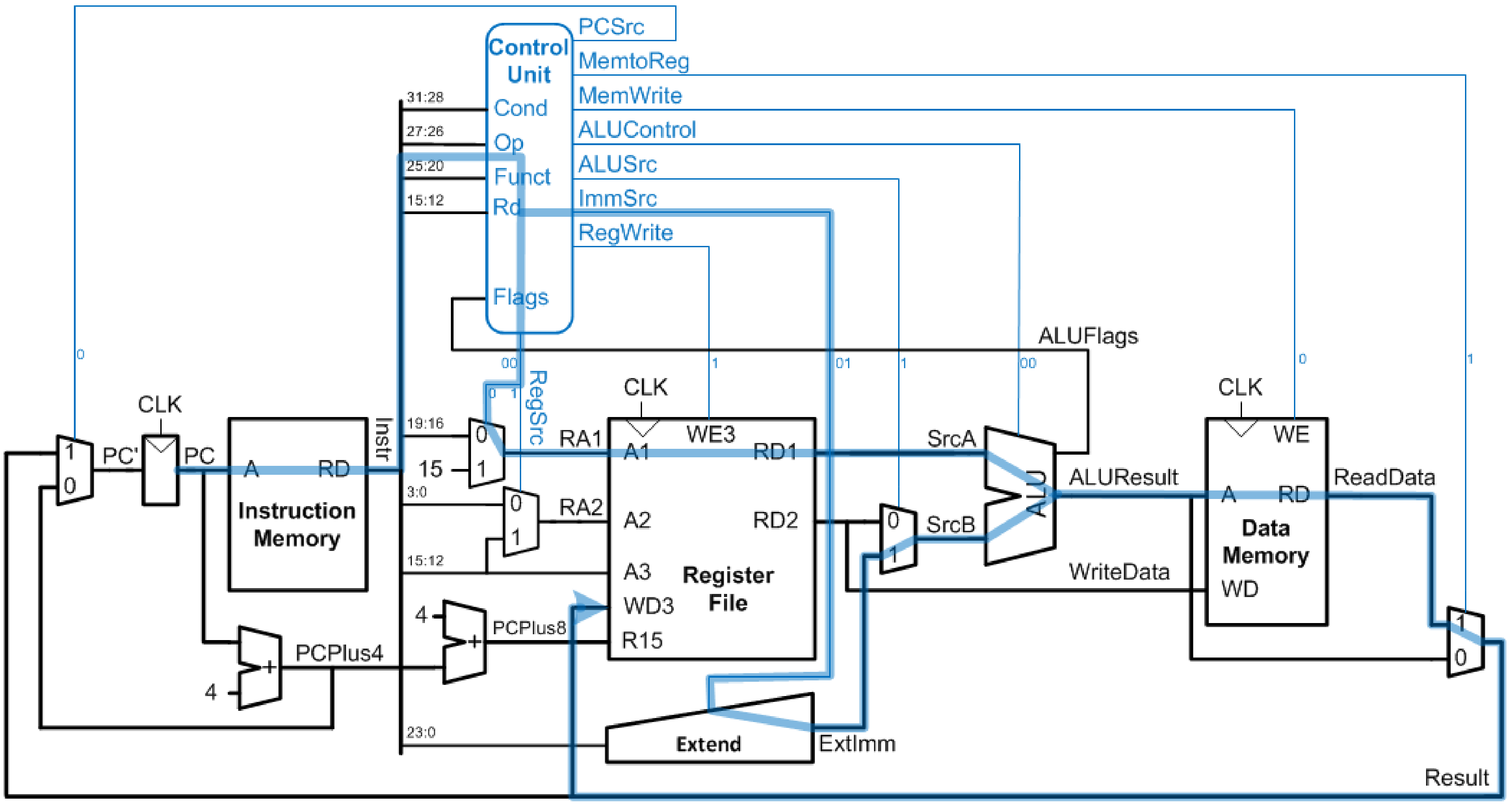


ADD R7, R2, R12, LSR #5

31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
14	0	0	4	0	2	7	5	01_2	0	12
cond	op	I	cmd	S	rn	rd	shamt5	sh		rm

- No change to the controller!

Single-Cycle Performance



- T_C (and hence, performance) limited by critical path (LDR)

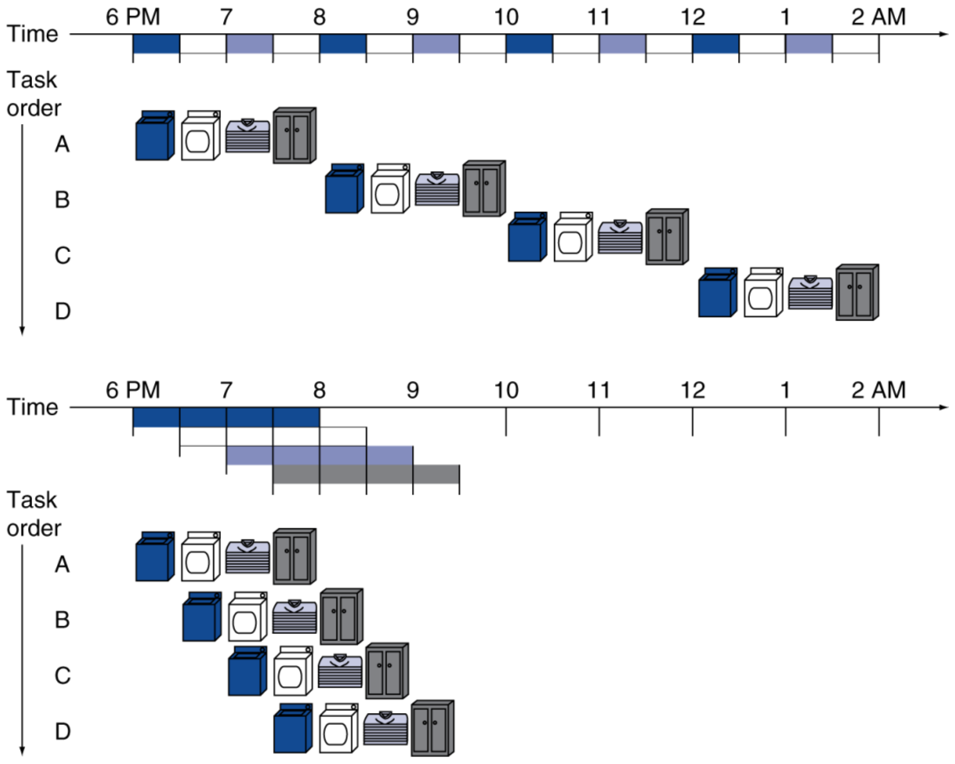
Single Cycle Design Summary

- Single-cycle - fetch, decode and execute each instructions in one clock cycle
  - (+) simple
  - (–) no datapath resource can be used more than once per instruction, so some must be duplicated
    - separate memories for instruction and data
    - 3 adders/ALUs
  - (–) cycle time limited by longest instruction (LDR)

- How Can We Make It Faster?
  - Pipelining (Chapter 6)
  - Superscalar (Chapter 7)
  - Multiprocessor systems (Chapter 8)

Pipelining Analogy. Stay Tuned for More!

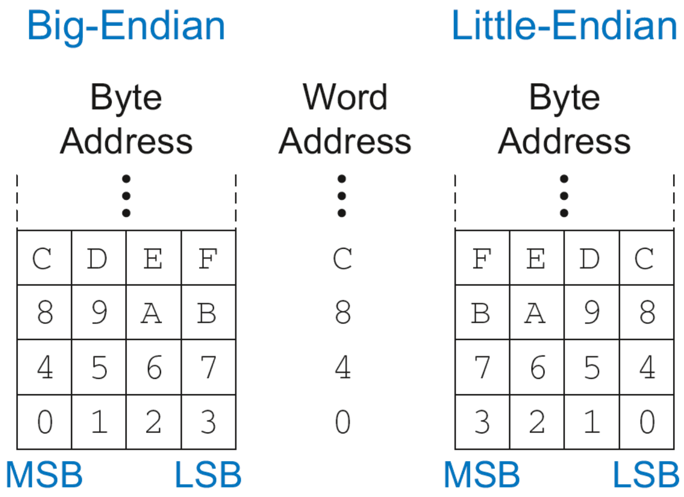
- Pipelined laundry: overlapping execution
  - Parallelism improves performance



- Four loads:
  - Speedup = 8/3.5 = 2.3
- Non-stop:
  - Speedup = 2n/(0.5n + 1.5) ≈ 4 = # of stages

Recap : Endianness

- Each data byte has unique address
- 32-bit word = 4 bytes, so word address increments by 4
- How to number bytes within a word?
  - Little-endian: byte numbers start at the little (least significant) end
  - Big-endian: byte numbers start at the big (most significant) end
  - It doesn’t really matter which addressing type is used
    – except when two systems share data



Big-Endian

Little-Endian

Byte Address				Word Address	Byte Address			
⋮				⋮	⋮			
C	D	E	F	C	F	E	D	C
8	9	A	B	8	B	A	9	8
4	5	6	7	4	7	6	5	4
0	1	2	3	0	3	2	1	0
MSB			LSB		MSB			LSB