

COMP2300-COMP6300-ENGN2219

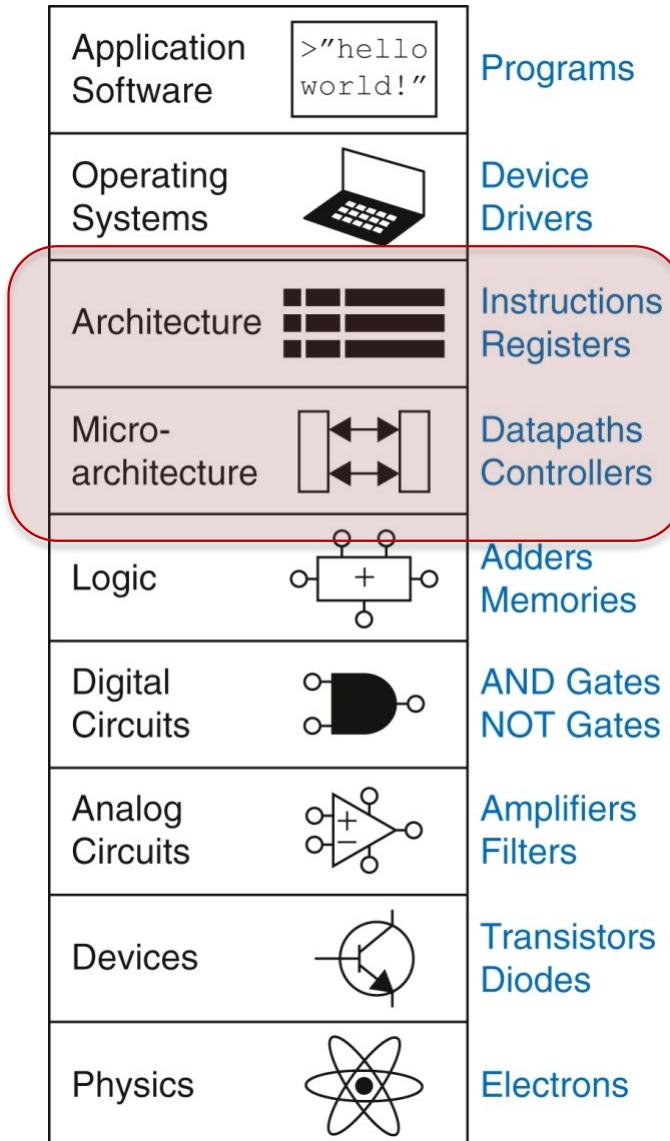
Computer Organization &

Program Execution

Convenor: Shoib Akram
shoib.akram@anu.edu.au



Australian
National
University



ISA then microarchitecture

Admin

- Assignment 1 is due on 11 April 2023, 10:00 am
- Inspiration for extension ideas CAN come from this week's lectures
 - Motivation for instructions, addressing modes, better microarchitecture
- Make use of office hours
 - Bring your questions ... opportunity to discuss anything

Branch Instructions

Program Counter (PC) points to
(contains the address of) next
instruction to execute

Byte Address	Instructions
.	.
.	.
.	.
0040000C	E 3 A 0 1 0 6 4
00400008	E 3 A 0 2 0 4 5
00400004	E 1 5 1 0 0 0 2
00400000	2 5 8 1 3 0 2 4
.	.
.	.

← PC

Normal (Sequential) Execution

- 32-Bit ISA with Byte-Addressable Memory
 - $PC = PC + 4$
- 64-Bit ISA with Byte-Addressable Memory
 - $PC = PC + 8$
- 32-Bit ISA with Word-Addressable Memory
 - $PC = PC + 1$

Normal (Sequential) Execution

Increment PC During Instruction
FETCH (to prepare to execute the
NEXT Instruction)

(1) Altering PC can break the sequential flow of program execution

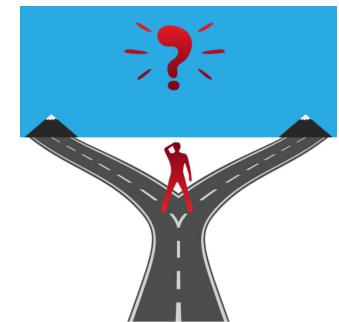
(2) Branch instructions alter the program counter **BUT** in a different way

Branch Instructions

- Typically, a computer program is executed in sequence
 - First instruction is executed, then the second, then the third, and so on
- **Decision making** is an important **advantage** of computers
 - `if` and `if-else` statements
 - `for` and `while` loops
 - `switch-case` statements
- ARM provides **branch** instructions to **skip** and **repeat** code

Type of Branches

- Branch (B)
 - Branches to another **TARGET** instruction
 - Unconditional branch: always executes the target instruction
 - Conditional branch: either executes the TARGET instruction or the next sequential instruction in memory based on a condition
 - **BEQ** (Branch if the Zero flag is set)
 - **BNE** (Branch if the Zero flag is not set)
- Branch and Link (BL)
 - A special branch to provide support for **functions** in C++ or Java
 - Architectural support for high-level language demands



Unconditional Branch

- The Branch in this example is **unconditional** and **always TAKEN (T)**

```
Assembly code:  
ADD R1, R2, #17  
B TARGET  
ORR R1, R1, R3  
AND R3, R1, #0xFF  
TARGET  
SUB R1, R1, #78
```

- After encountering **B**, the CPU executes **SUB** instead of **ORR**
- The **label TARGET** is a **memory address** in human readable form
 - TARGET** is transformed into a **memory address** by a tool called **assembler**
 - Assemblers transform assembly code into machine code (0's and 1's)

Assembly language let us give meaningful
(human-readable and easy to differentiate)
symbolic names (labels) to memory locations,
such as TARGET, rather than use binary addresses

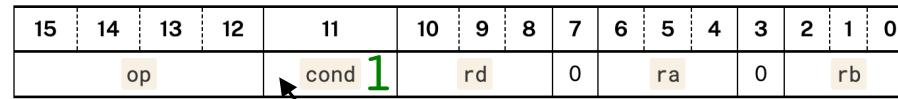
We call these names **Symbolic Addresses**

Conditional Branch

- Conditional branch uses condition mnemonics
- Let's revisit conditional execution and condition mnemonics

Recall: Conditional Execution in QuAC

- Bit 11 is associated with a **condition code**



- **addeq r1, r2, r3 (cond = TRUE)**
- If **cond field** (Instr_{11}) is **TRUE**, then
 - Execute the instruction only if the last ALU instruction set the **Z** flag to **TRUE**
 - Otherwise, do not execute the instruction (**depart from the usual control flow**)
- What is the relationship between **eq** and **Z** flag?

Name	Suffix	Encoding	Condition	Meaning
Always	-	0	-	Always executes
Equals	eq	1	Z == 1	Execute if latest ALU result was zero

Recall: ARM Condition Mnemonics

<i>cond</i>	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	$Z \text{ OR } \bar{C}$
1010	GE	Signed greater than or equal	$N \oplus V$
1011	LT	Signed less than	$N \oplus \bar{V}$
1100	GT	Signed greater than	$\bar{Z}(N \oplus V)$
1101	LE	Signed less than or equal	$Z \text{ OR } (N \oplus V)$
1110	AL (or none)	Always / unconditional	ignored

Example: Do it Yourself

Page 308 of H&H

- R2 = 0x80000000 and R3 = 0x00000001
- Which instructions will execute?
 - Flags: N Z C V = ?

CMP	R2,	R3	
ADDEQ	R4,	R5,	#78
ANDHS	R7,	R8,	R9
ORRMI	R10,	R11,	R12
EORLT	R12,	R7,	R10

Conditional Branch

- Conditional branch uses **condition mnemonics**

Assembly code:

```
MOV R0, #4
ADD R1, R0, R0
CMP R0, R1
BEQ THERE
ORR R1, R1, R1
THERE
ADD R1, R1, #78
```

- CMP subtracts **R1** from **R0** and **sets** all **flags**
 - Z flag is **FALSE** because $R0 - R1$ is not 0
- The branch **BEQ** evaluates to **FALSE**
 - Branch is **NOT TAKEN (NT)**
 - The next instruction executed is the ORR instruction

Instruction Format – 3: Branch

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

- **op = 10**
- **imm24 = 24-bit signed immediate**
- The two bits [25:24] form the funct field
- - Bit 25 is always **1**
 - **L bit:** L = **0** for **B** (**Branch**)
 - **L bit:** L = **1** for **BL** (**Branch and Link**)
- Format
 - B **TARGET**
 - B **imm24**

Branch with L = 0

- Branch with L bit (Bit 24) as 0 is a regular branch

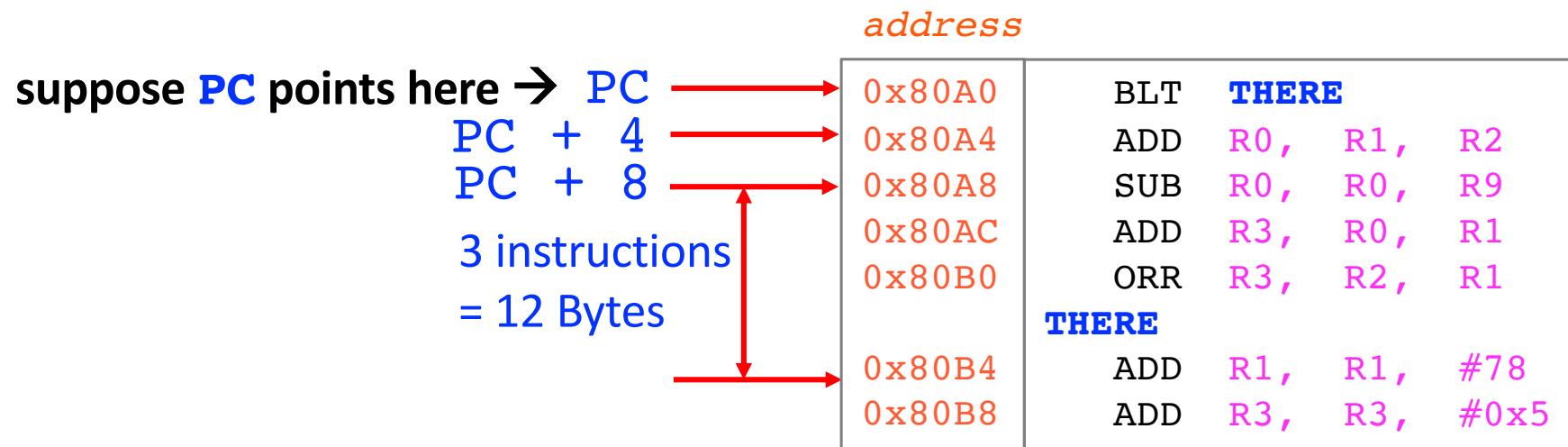
31:28	27:26	25:24	23:0
cond	10	10	imm24

- **Branch Target Address (BTA)**: The address of the next instruction to execute if the branch is taken
- How is BTA calculated?
 1. Shift left **imm24** by 2 (to convert **words to bytes**)
 2. Sign-extend (copy **Instruction[25]** into **Instruction[26:31]**)
 3. Add **PC + 8**

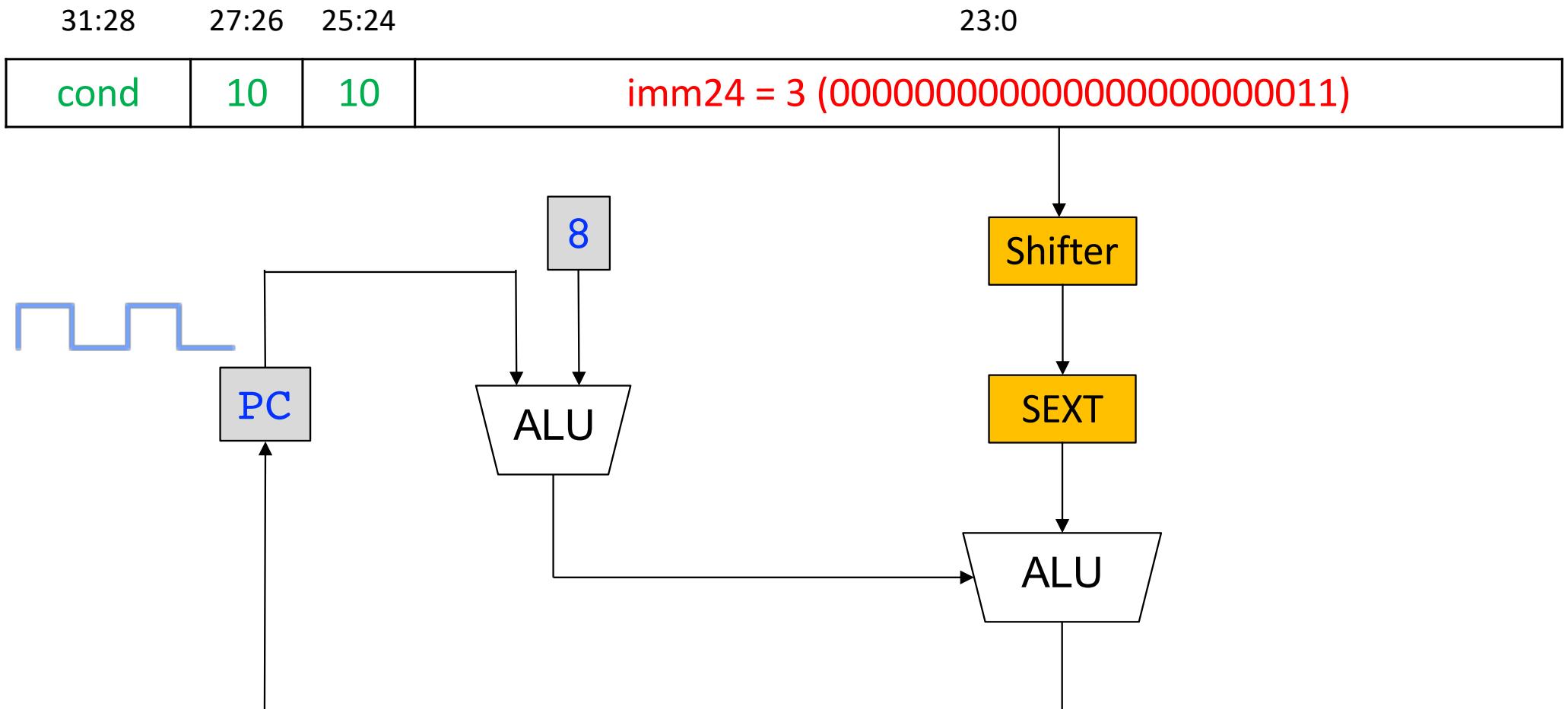
BTA Calculation Example

- Instruction encodes the distance from PC + 8 as 3 32-bit words

31:28	27:26	25:24	23:0
cond	10	10	imm24 = 3 (0000000000000000000000000011)



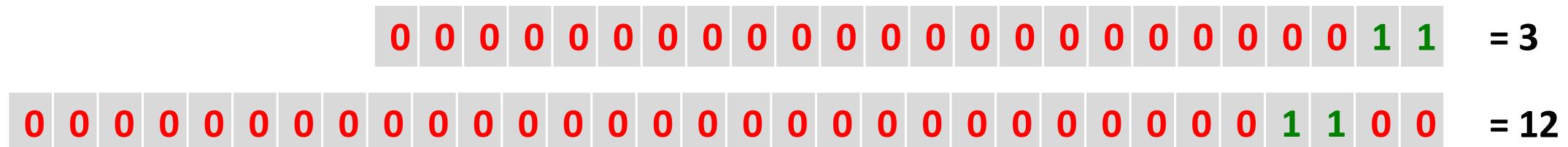
BTA Calculation DataPath



BTA Calculation Summary

The processor calculates the **BTA** in three steps

1. Shift left imm24 by 2 (to convert words to bytes)
2. Sign-extend (copy Instr_{25} into $\text{Instr}_{31:26}$)
3. Add PC + 8



Branch-Related Terminology

- Two main types of branches
 - Conditional branch: Executes the next sequential instruction or TARGET instruction based on a condition
 - Unconditional branch: Always (unconditionally) executes the TARGET instruction
- Branch Target
 - Memory address of the TARGET instruction
- Branch Condition
 - Condition which if TRUE branch jumps to the TARGET instruction
- Branch Resolution/Evaluation
 - The act of evaluating the branch condition
 - Two outcomes of branch resolution are:
 - Taken Branch (T): branch condition evaluates to TRUE
 - Untaken (Not Taken or NT) Branch: branch evaluates to FALSE
- Branch behavior
 - Strongly (most of times) Taken/Untaken OR Weakly (some of the times) Taken/Untaken
 - Always Taken OR Always Untaken
- Branch Prediction
 - In pipelined CPUS, branch resolves in EXECUTE stage while the TARGET is needed in FETCH stage
 - Modern CPUs use a branch predictor to predict the branch direction (T/NT) and branch TARGET

if and if-else

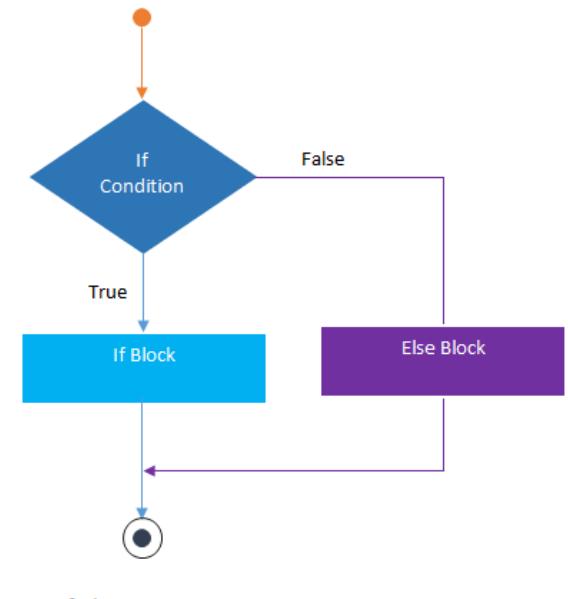
- We will study high-level language (C) to assembly transformation in this course

The Three Program Constructs

- We will see three basic constructs used in **structured programs** (**construct** comes from **constructing** a program)
- Sequential ✓
 - One subtask, followed by other, never going back to first
- Conditional
 - One of the two subtasks but not both
- Iterative
 - Doing a subtask a number of times

Conditional Statements

- If the condition is **TRUE**, do one subtask, otherwise, do a different subtask
- A subtask may do nothing
- We call it a conditional construct
- All languages provides conditional constructs
- Subtask = Block of code (instructions)



if Statement

C code:

```
if (apples == oranges)
    f = i + 1;
f = f - i;
```

Assembly code:

```
; R0 = apples
; R1 = oranges
; R2 = f
; R3 = i
    CMP R0, R1
    BNE L1
    ADD R2, R3, #1
L1
    SUB R2, R2, R3
```

- apples == oranges?
- if yes, branch to L1
- if no, add 1 to i
- Subtract i from f

- The assembly code checks for the **opposite condition** in C code
- Skips the **if** block if the **condition** is not satisfied
- If the branch is **NOT TAKEN**, the **if** block is executed

if Statement

- It is very rarely the case that computer programs can be written only one way
- (Try it yourself) Use the **BEQ** instruction instead of **BNE**
 - Using conditional execution (**next**)

if with Conditional Execution

C code:

```
if (apples == oranges)
    f = i + 1;
f = f - i;
```



Assembly code:

```
; R0 = apples
; R1 = oranges
; R2 = f
; R3 = i
    CMP    R0,   R1
    ADDEQ R2,   R3,   #1
    SUB    R2,   R2,   R3
```

- apples == oranges?
- if yes, add 1 to i
- Subtract i from f

- This solution is **shorter** and **faster** (**one fewer instruction**)
- If the **if** block is long, **it is tedious to write** conditional mnemonics
- Conditional execution requires **NEEDLESS fetching of instructions** from memory
- In **pipelined CPUs**, **branch instructions introduce extra delay**
 - Think: If the branch predictor **makes a mistake** (**branch misprediction**)

if-else Statement

C code:

```
if (apples == oranges)
    f = i + 1;
else
    f = f - i;
```

Assembly code:

```
; R0 = apples
; R1 = oranges
; R2 = f
; R3 = i
    CMP    R0,   R1
    BNE    L1
    ADD    R2,   R3,   #1
    B      L2
L1
    SUB    R2,   R2,   R3
L2
```

- apples == oranges?
- if yes, branch to L1
- if no, add 1 to i
- Branch to L2
- Subtract i from f

if-else Statement

C code:

```
if (apples == oranges)
    f = i + 1;
else
    f = f - i;
...
```

Assembly code:

```
; R0 = apples
; R1 = oranges
; R2 = f
; R3 = i
CMP  R0,  R1
BNE  L1
ADD  R2,  R3,  #1
B    L2
L1
SUB  R2,  R2,  R3
L2
```

- apples == oranges?
- if yes, branch to L1
- if no, add 1 to i
- Branch to L2
- Subtract i from f

if-else Statement

C code:

```
if (apples == oranges)
    f = i + 1;
else
    f = f - i;
...
```

Assembly code:

```
; R0 = apples
; R1 = oranges
; R2 = f
; R3 = i
CMP R0, R1
BNE L1
ADD R2, R3, #1
B L2
L1
SUB R2, R2, R3
L2
```

- apples == oranges?
- if yes, branch to L1
- if no, add 1 to i
- Branch to L2
- Subtract i from f

if-else Statement

C code:

```
if (apples == oranges)
    f = i + 1;
else
    f = f - i;
...
```

Assembly code:

```
; R0 = apples
; R1 = oranges
; R2 = f
; R3 = i
    CMP R0, R1
    BNE L1
    ADD R2, R3, #1
    B L2
L1
    SUB R2, R2, R3
L2
```

- apples == oranges?
- if yes, branch to L1
- if no, add 1 to i
- Branch to L2
- Subtract i from f

if-else Statement

C code:

```
if (apples == oranges)
    f = i + 1;
else
    f = f - i;
...
```

Assembly code:

```
; R0 = apples
; R1 = oranges
; R2 = f
; R3 = i
    CMP    R0,   R1
    BNE    L1
    ADD    R2,   R3,   #1
    B      L2
L1
    SUB    R2,   R2,   R3
L2
```

- apples == oranges?
- if yes, branch to L1
- if no, add 1 to i
- Branch to L2
- Subtract i from f

if-else Statement

C code:

```
if (apples == oranges)
    f = i + 1;
else
    f = f - i;
...
```

Assembly code:

```
; R0 = apples
; R1 = oranges
; R2 = f
; R3 = i
    CMP R0, R1
    BNE L1
    ADD R2, R3, #1
    B L2
L1
    SUB R2, R2, R3
L2
```

- apples == oranges?
- if yes, branch to L1
- if no, add 1 to i
- Branch to L2
- Subtract i from f

if-else Statement

C code:

```
if (apples == oranges)
    f = i + 1;
else
    f = f - i;
...
```

Assembly code:

```
; R0 = apples
; R1 = oranges
; R2 = f
; R3 = i
CMP  R0,  R1
BNE  L1
ADD  R2,  R3,  #1
B    L2
L1
SUB  R2,  R2,  R3
L2
```

- apples == oranges?
- if yes, branch to L1
- if no, add 1 to i
- Branch to L2
- Subtract i from f

if-else Statement

- It is very rarely the case that computer programs can be written only one way
- **Do it yourself:** Find an alternative way to write the if-else statement

if-else with Conditional Execution

C code:

```
if (apples == oranges)
    f = i + 1;
else
    f = f - i;
```

Assembly code:

```
; R0 = apples
; R1 = oranges
; R2 = f
; R3 = i
CMP    R0, R1
ADDEQ R2, R3, #1
SUBNE R2, R2, R3
```

- This solution is **shorter** and **faster** (one fewer instruction)
- If the **if** block is long, **it is tedious to write** conditional mnemonics
- Conditional execution requires **NEEDLESS** fetching of instructions from memory
- In **pipelined** CPUs, **branch instructions introduce extra delay**
 - Think: If the branch predictor **makes a mistake** (**branch misprediction**)

Switch Statement

switch-case Statement

```
C code:  
switch (button) {  
    case 1: atm = 20; break;  
    case 2: atm = 50; break;  
    case 3: atm = 100; break;  
    default: atm = 0; break;  
}
```

- Execute one of several blocks of code (**cases**) depending on the condition
-
- **Break** out of the entire **switch** block {...} after executing a specific case (block)
- In the above example condition is the state of variable **button**
- If no conditions are met, the **default** block is executed

switch-case Statement

C code:

```
switch (button) {  
    case 1: atm = 20; break;  
    case 2: atm = 50; break;  
    case 3: atm = 100; break;  
    default: atm = 0; break;  
}
```

Assembly code:

```
; R0 = button  
; R1 = atm  
    CMP    R0, #1  
    MOVEQ  R1, #20  
    BEQ    DONE  


---

    CMP    R0, #2  
    MOVEQ  R1, #50  
    BEQ    DONE  


---

    CMP    R0, #3  
    MOVEQ  R1, #100  
    BEQ    DONE  


---

    MOV    R1, #0  
DONE
```

- Comment begins with ;
- Another comment
- is button == 1?
- atm = 20
- break out
- is button == 2?
- atm = 50
- break out
- is button == 3?
- atm = 100
- break out
- Execute default case

switch-case Statement

C code:

```
switch (button) {  
    case 1: atm = 20; break;  
    case 2: atm = 50; break;  
    case 3: atm = 100; break;  
    default: atm = 0; break;  
}
```

Assembly code:

```
; R0 = button  
; R1 = atm  
  
    CMP    R0, #1  
    MOVEQ  R1, #20  
    BEQ    DONE  
  
    CMP    R0, #2  
    MOVEQ  R1, #50  
    BEQ    DONE  
  
    CMP    R0, #3  
    MOVEQ  R1, #100  
    BEQ    DONE  
  
    MOV    R1, #0  
  
    DONE
```

- Comment begins with ;
- Another comment
- is button == 1?
- atm = 20
- break out
- is button == 2?
- atm = 50
- break out
- is button == 3?
- atm = 100
- break out
- Execute default case

switch-case Statement

C code:

```
switch (button) {  
    case 1: atm = 20; break;  
    case 2: atm = 50; break;  
    case 3: atm = 100; break;  
    default: atm = 0; break;  
}
```

Assembly code:

```
; R0 = button  
; R1 = atm  
    CMP    R0, #1  
    MOVEQ  R1, #20  
    BEQ    DONE  
  
    CMP    R0, #2  
    MOVEQ  R1, #50  
    BEQ    DONE  
  
    CMP    R0, #3  
    MOVEQ  R1, #100  
    BEQ    DONE  
  
    MOV    R1, #0
```

DONE

- Comment begins with ;
- Another comment
- is button == 1?
- atm = 20
- break out
- is button == 2?
- atm = 50
- break out
- is button == 3?
- atm = 100
- break out
- Execute default case

switch-case Statement

C code:

```
switch (button) {  
    case 1: atm = 20; break;  
    case 2: atm = 50; break;  
    case 3: atm = 100; break;  
    default: atm = 0; break;  
}
```

Assembly code:

```
; R0 = button  
; R1 = atm  
    CMP    R0, #1  
    MOVEQ  R1, #20  
    BEQ    DONE  
  
    CMP    R0, #2  
    MOVEQ  R1, #50  
    BEQ    DONE  
  
    CMP    R0, #3  
    MOVEQ  R1, #100  
    BEQ    DONE  
    MOV    R1, #0  
  
DONE
```

- Comment begins with ;
- Another comment
- is button == 1?
- atm = 20
- break out
- is button == 2?
- atm = 50
- break out
- is button == 3?
- atm = 100
- break out
- Execute default case

switch-case Statement

C code:

```
switch (button) {  
    case 1: atm = 20; break;  
    case 2: atm = 50; break;  
    case 3: atm = 100; break;  
    default: atm = 0; break;  
}
```

Assembly code:

```
; R0 = button  
; R1 = atm  
    CMP    R0, #1  
    MOVEQ  R1, #20  
    BEQ    DONE  
  
    CMP    R0, #2  
    MOVEQ  R1, #50  
    BEQ    DONE  
  
    CMP    R0, #3  
    MOVEQ  R1, #100  
    BEQ    DONE  
  
    MOV    R1, #0  
  
DONE
```

- Comment begins with ;
- Another comment
- is button == 1?
- atm = 20
- break out
- is button == 2?
- atm = 50
- break out
- is button == 3?
- atm = 100
- break out
- Execute default case

Shift Instructions

Category: Data Processing

Shift Instructions

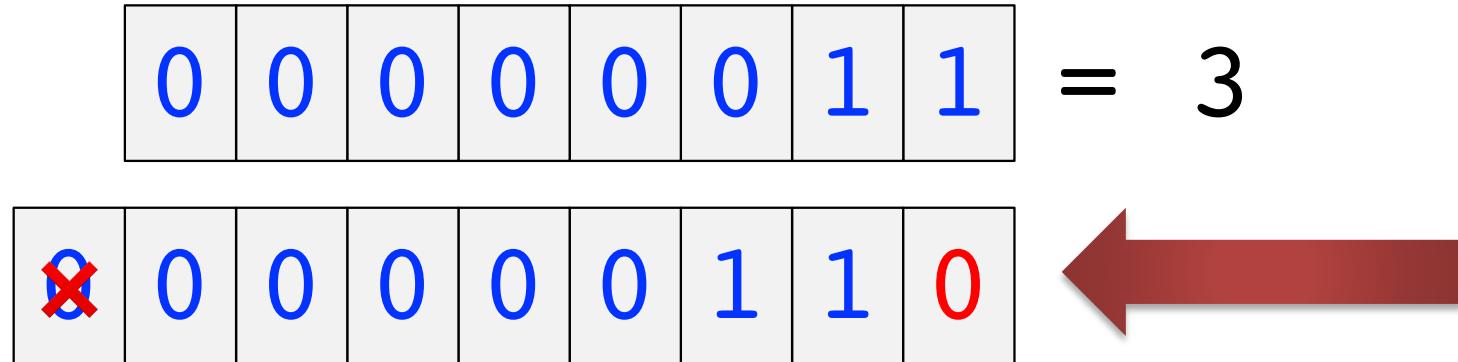
- Shift the value in a register left or right, drop bits off the end
 - Logical Shift Left (**LSL**)
 - Logical Shift Right (**LSR**)
 - Arithmetic Shift Right (**ASR**)
 - Rotate Right (**ROR**)
- **Logical Shift:** shifts the number to the left/right and fills empty slots with zero
- **Arithmetic Shift:** On right shifts fill the most significant bits with the sign bit
- **Rotate:** rotates number in a circle such that empty spots are filled with bits shifted off the other end

Logical Shift Left (LSL)

0	0	0	0	0	0	1	1
---	---	---	---	---	---	---	---

- Binary Number in Decimal = 3

Logical Shift Left (LSL)



- Shift the number **LEFT** by **ONE BIT**
- **INSERT 0** in **Least Significant Position**
- Get **RID** of the **Most Significant BIT**

Logical Shift Left (LSL)

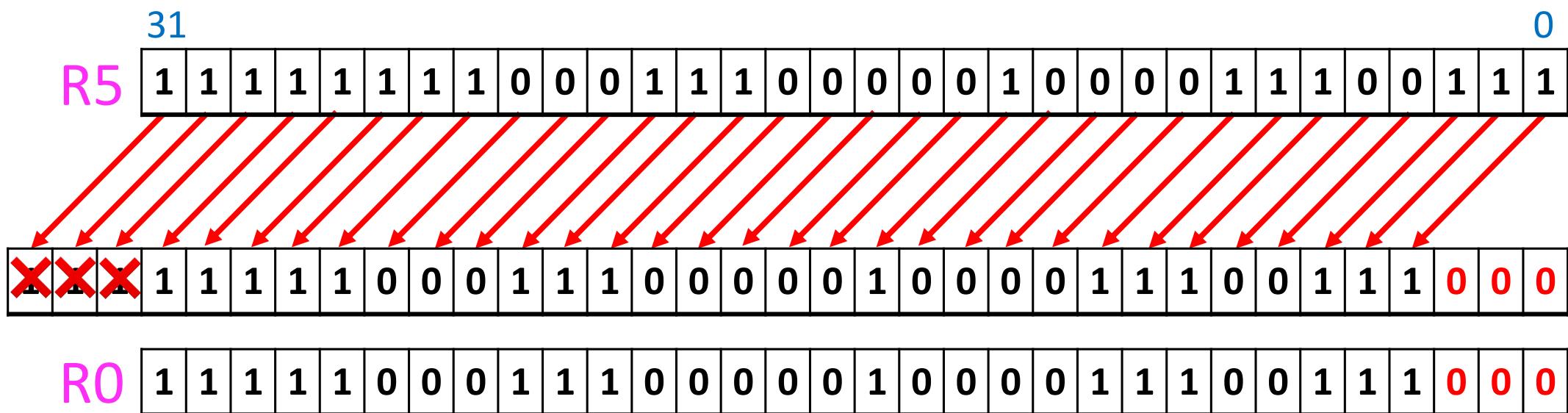
0	0	0	0	0	0	1	1	= 3
0	0	0	0	0	1	1	0	= 6

- Binary Number after shift in Decimal = 6
- SHIFT LEFT = MULTIPLY BY 2

Logical Shift Left (LSL)

ARM Instruction

LSL R0, R5, #3

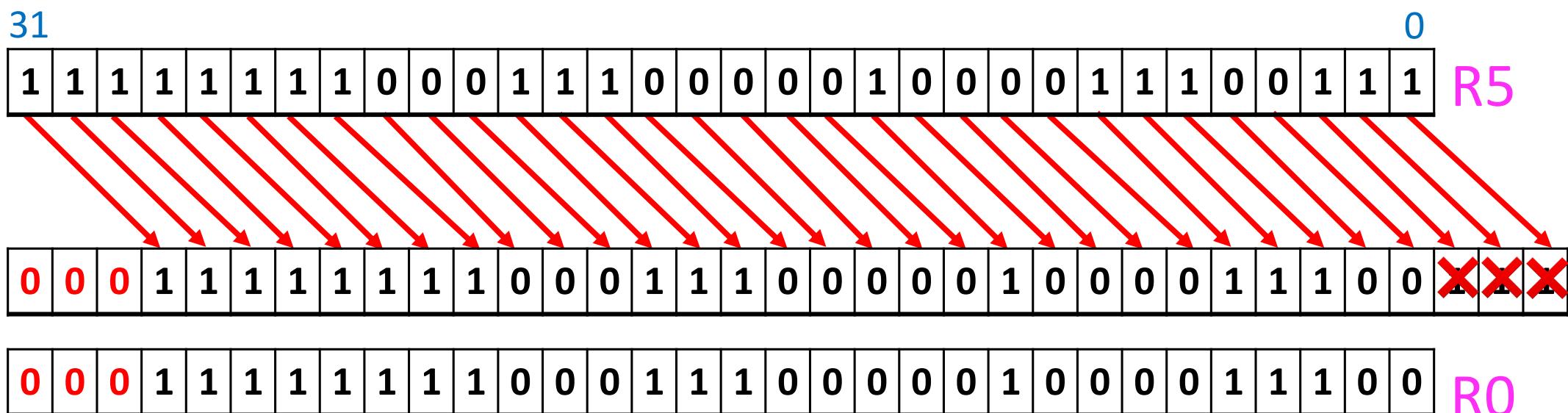


- Shift all bits left **3** positions, fill **3** least significant bits with **0's**
- Drop the **3** bits off the end

Logical Shift Right (LSR)

ARM Instruction

LSR R0, R5, #3

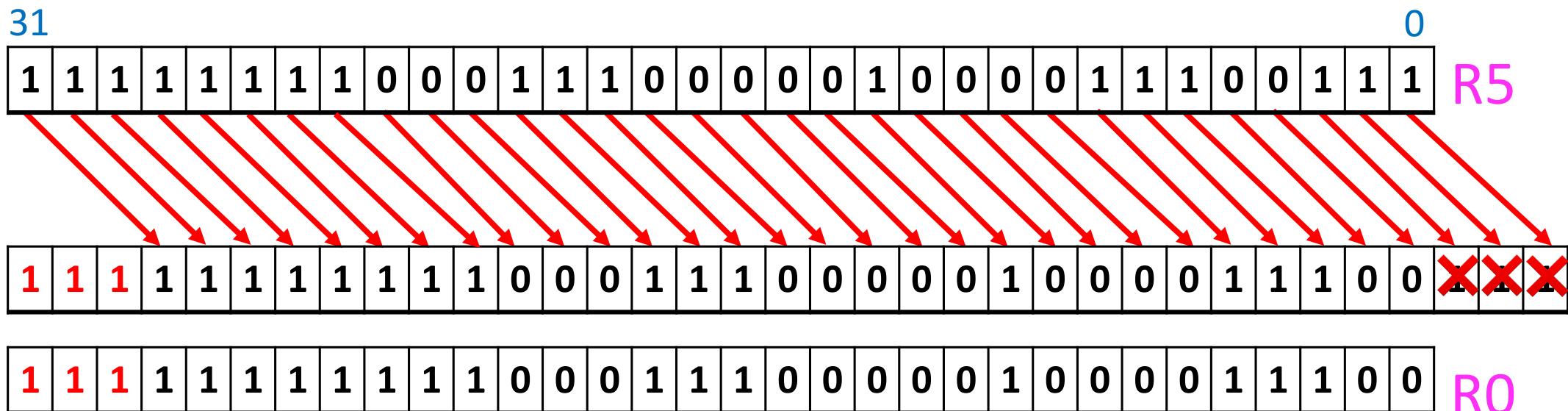


- Shift all bits right 3 positions, insert three 0's from the right
- Drop the 3 bits from the left

Arithmetic Shift Right (LSR)

ARM Instruction

ASR R0, R5, #3

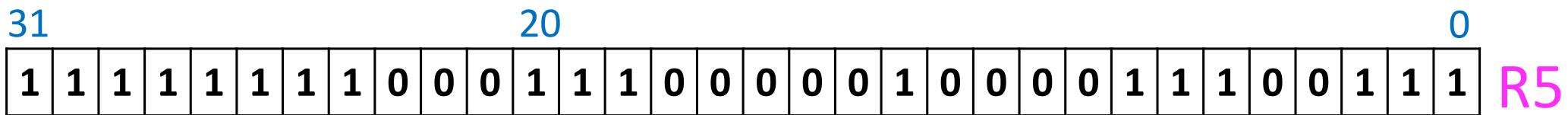


- Shift all bits right 3 positions, insert three 1's from the right
- Drop the 3 bits from the left

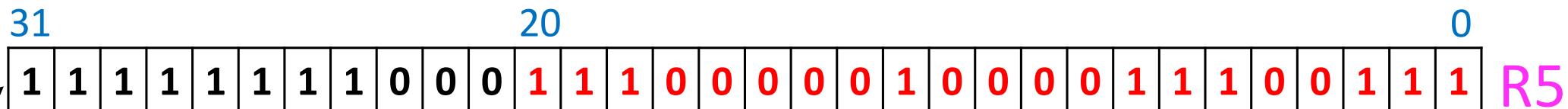
Rotate Right (ROR)

ARM Instruction

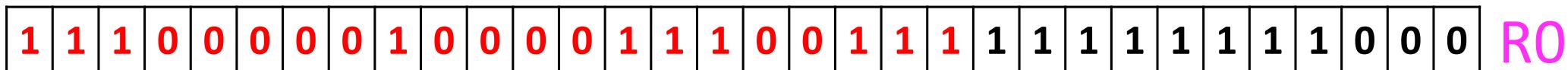
```
ROR R0, R5, #21
```



- Do a circular shift
- Right shift by 21 and put back bits that fall off at left end



Result



Binary Encoding of Shift Instructions

- Self Study
- Section 6.4 of H&H

Shifts: Machine Representation

	31:28	27:26	25	24:21	20	19:16	15:12	11:4	3:0
DP-R	cond	00	0	cmd	S	Rn	Rd	0 0 0 0 0 0 0 0	Rm

Shift Instructions

	31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
	cond	00	0	1101	S	0000	Rd	shamt5	sh	0	Rm

- cmd = 1101
- sh = 00 (LSL), 01 (LSR), 10 (ASR), 11 (ROR)
- Rn = 0
- sham5 = 5-bit shift amount

Shifts: Machine Representation

- Format (Src2 = Register)

LSL R0, R5, #3
↓ ↓ ↓
LSL Rd, Rm, shamt5

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

Shifts: Machine Representation

- ARM also has instructions with shift amount held in a register

LSL R4, R8, R6

ROR R5, R8, R6

31:28	27:26	25	24:21	20	19:16	15:12	11:8	7	6:5	4	3:0
cond	00	0	cmd	S	Rn	Rd	Rs	0	sh	1	Rm

Use of Shift Instructions

- **Left shift by N** = Multiplication by 2^N
- **Arithmetic right shift by N** = Division by 2^N
- Extract bits or assemble new bit patterns
 - Network programming
 - Cryptography
 - Compression of data

Examples of Shift Instructions

Source register			
R5	1111 1111	0001 1100	0001 0000
Assembly Code		Result	
LSL R0, R5, #7 R0		1000 1110	0000 1000
LSR R1, R5, #17 R1		0000 0000	0000 0000
ASR R2, R5, #3 R2		1111 1111	1110 0011
ROR R3, R5, #21 R3		1110 0000	1000 0111

Figure 6.4 Shift instructions with immediate shift amounts

Source registers			
R8	0000 1000	0001 1100	0001 0110
R6	0000 0000	0000 0000	0000 0000
Assembly code		Result	
LSL R4, R8, R6 R4		0110 1110	0111 0000
ROR R5, R8, R6 R5		1100 0001	0110 1110

Figure 6.5 Shift instructions with register shift amounts

Shift amount can be in a register

For Loop

Loops

- Life is full of repetition!
 - Standard routines repeat each day, week, month, ...
 - Terminating at some point
- Repetition (Iteration) is also the essence of computing!
 - Compute the sum of first one billion numbers
 - Go over each student record and change numerical grade to letter
 - Terminate if no more records are found
- Looping sometimes but not always (depending on a condition) is where a computer rocks!

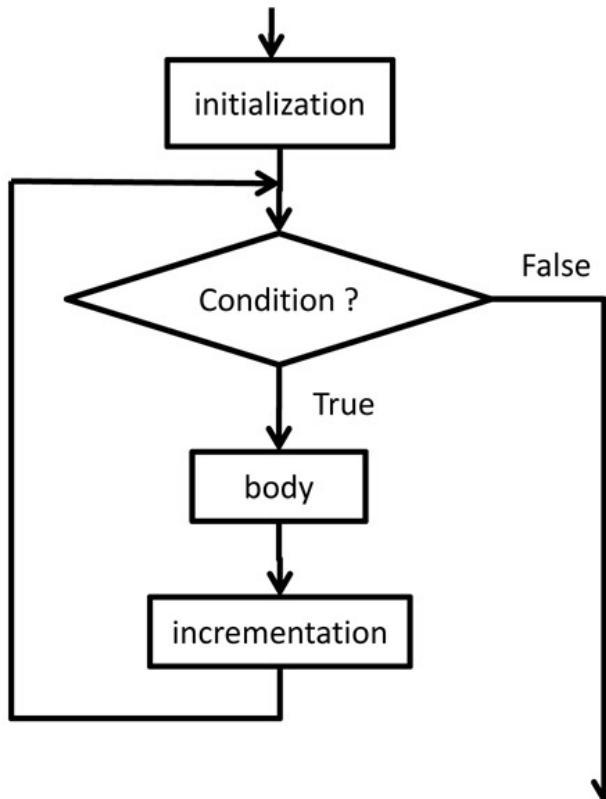
Loops

- Loops are **iterative** constructs that repeat a subtask several times, but only as long as some **condition** is **TRUE** (**subtask** = sequence of instructions)
- If the condition is **TRUE**, do the **subtask** (also called **loop body**)
- After the subtask is finished, go back and check the **condition** again
- As long as the result of the condition is **TRUE**, the program continues to carry out the same subtask again and again
- The first time the test is **NOT TRUE**, the program proceeds onward

Loops

- Loops are **iterative**, only as long as so
- If the condition is
- After the subtask
- As long as the result carry out the same
- The first time the

```
for( initialization; condition; incrementation )  
    body;
```



real times, but

ody)

tion again

continues to

onward

Loops

- We will look at
 - For Loops
 - While Loop
- Our focus
 - How are loops in high-level languages transformed (translated) into assembly by human or compiler?

For Loop in C

C code:

```
int i;
int sum = 0;

for (i = 0; i < 10; i = i + 1)
    sum = sum + i;
...
...
```

- The variable “**i**” is called the loop **index** or **counter**
- The **For statement** has three components
 - **i = 0** : index initialization
 - **i < 10** : loop termination condition
 - **i = i + 1** : loop advancement
- The **body** of the loop can have one or more statements

For Loop in ARM Assembly

C code:

```
int i;  
int sum = 0;  
  
for (i = 0; i < 10; i = i + 1)  
    sum = sum + i;  
...  
...
```

Assembly code:

```
; R0 = i  
; R1 = sum  
MOV R0, #0  
MOV R1, #0  
FOR  
    CMP R0, #10  
    BGE DONE  
    ADD R1, R1, R0  
    ADD R0, R0, #1  
    B FOR  
DONE
```

- Comment begins with ;
- Another comment
- Initialize i
- Initialize sum
- Label/Address of CMP
- check condition: $i < 10$?
- if ($i \geq 10$) exit loop
- $sum = sum + i$
- Increment i
- repeat loop

- **High-level code:** Few lines (**statements**); **Assembly code:** Many lines (**instructions**)
- **High-level code:** Variable names; **Assembly code:** Registers & memory addresses
- **High-level code:** Hides machine details (e.g., **MOV**ement); **ASM:** Expose details
- In both C and assembly, the **control flow** (**sequential** and **iterative** constructs) are visible
 - Easier to identify in C, more difficult in assembly
- Let's do a line-by-line comparison of the above code ...

For Loop in ARM Assembly

C code:

```
int i;  
int sum = 0;  
  
for (i = 0; i < 10; i = i + 1)  
    sum = sum + i;  
...  
...
```

Assembly code:

```
; R0 = i  
; R1 = sum  
MOV R0, #0  
MOV R1, #0  
FOR  
CMP R0, #10  
BGE DONE  
ADD R1, R1, R0  
ADD R0, R0, #1  
B FOR  
DONE
```

- Comment begins with ;
- Another comment
- Initialize i
- Initialize sum
- Label/Address of CMP
- check condition: $i < 10$?
- if ($i \geq 10$) exit loop
- $sum = sum + i$
- Increment i
- repeat loop

- In high-level language programs, we **initialize variables**
 - In assembly **initializing variables** translates to **initializing registers**

For Loop in ARM Assembly

C code:

```
int i;  
int sum = 0;  
  
for (i = 0; i < 10; i = i + 1)  
    sum = sum + i;  
...  
...
```

Assembly code:

```
; R0 = i  
; R1 = sum  
MOV R0, #0  
MOV R1, #0  
FOR  
    CMP R0, #10  
    BGE DONE  
    ADD R1, R1, R0  
    ADD R0, R0, #1  
    B FOR  
DONE
```

- Comment begins with ;
- Another comment
- Initialize i
- Initialize sum
- Label/Address of CMP
- check condition: $i < 10$?
- if ($i \geq 10$) exit loop
- $sum = sum + i$
- Increment i
- repeat loop

For Loop in ARM Assembly

C code:

```
int i;  
int sum = 0;  
  
for (i = 0; i < 10; i = i + 1)  
    sum = sum + i;  
...  
...
```

Assembly code:

```
; R0 = i  
; R1 = sum  
MOV R0, #0  
MOV R1, #0  
FOR  
    CMP R0, #10  
    BGE DONE  
    ADD R1, R1, R0  
    ADD R0, R0, #1  
    B FOR  
DONE
```

- Comment begins with ;
- Another comment
- Initialize i
- Initialize sum
- Label/Address of CMP
- check condition: $i < 10$?
- if ($i \geq 10$) exit loop
- $sum = sum + i$
- Increment i
- repeat loop

- Check **termination** condition to break out of the loop if condition is met

For Loop in ARM Assembly

C code:

```
int i;  
int sum = 0;  
  
for (i = 0; i < 10; i = i + 1)  
    sum = sum + i;  
...  
...
```

Assembly code:

```
; R0 = i  
; R1 = sum  
MOV R0, #0  
MOV R1, #0  
FOR  
    CMP R0, #10  
    BGE DONE  
    ADD R1, R1, R0  
    ADD R0, R0, #1  
    B FOR  
DONE
```

- Comment begins with ;
- Another comment
- Initialize i
- Initialize sum
- Label/Address of CMP
- check condition: $i < 10$?
- if ($i \geq 10$) exit loop
- $sum = sum + i$
- Increment i
- repeat loop

- Add the loop counter i to the variable sum

For Loop in ARM Assembly

C code:

```
int i;  
int sum = 0;  
  
for (i = 0; i < 10; i = i + 1)  
    sum = sum + i;  
...  
...
```

Assembly code:

```
; R0 = i  
; R1 = sum  
MOV R0, #0  
MOV R1, #0  
FOR  
    CMP R0, #10  
    BGE DONE  
    ADD R1, R1, R0  
    ADD R0, R0, #1  
    B FOR  
DONE
```

- Comment begins with ;
- Another comment
- Initialize i
- Initialize sum
- Label/Address of CMP
- check condition: $i < 10$?
- if ($i \geq 10$) exit loop
- $sum = sum + i$
- Increment i
- repeat loop

- Increment the loop counter

For Loop in ARM Assembly

C code:

```
int i;  
int sum = 0;  
  
for (i = 0; i < 10; i = i + 1)  
    sum = sum + i;  
...  
...
```

Assembly code:

```
; R0 = i  
; R1 = sum  
MOV R0, #0  
MOV R1, #0  
FOR  
    CMP R0, #10  
    BGE DONE  
    ADD R1, R1, R0  
    ADD R0, R0, #1  
    B FOR  
DONE
```

- Comment begins with ;
- Another comment
- Initialize i
- Initialize sum
- Label/Address of CMP
- check condition: $i < 10$?
- if ($i \geq 10$) exit loop
- $sum = sum + i$
- Increment i
- repeat loop

- Keep iterating by branching back to the CMP instruction

Same **For** Loop in a Different Style

C code:

```
int i;  
int sum = 0;  
  
for (i = 0; i < 10; i = i + 1)  
    sum = sum + i;
```

Assembly code:

```
; R0 = i  
; R1 = sum  
MOV R0, #0  
MOV R1, #0  
COND  
    CMP R0, #10  
    BLT FOR  
    B DONE  
FOR  
    ADD R1, R1, R0  
    ADD R0, R0, #1  
    B COND  
DONE
```

- check condition
- if $i < 10$ repeat
- if $i \geq 10$, leave for
- add sum to i
- Increment i
- Iterate again

- More faithfully follows the for loop semantics in C
- Use **BLT** instead of **BGE**
- Different ways to translate a high-level statement into ASM

Aside: Syntax versus Semantics

- **Syntax:** Arrangement of keywords in a statement
 - There is a ; after a statement
 - The loop statement uses parentheses
- **Semantics:** Meaning of keywords and their arrangement
 - Repeat the instructions in the loop body until condition is not met
 - Add **sum** to **i**
 - What the CPU does depends on statement and instruction semantics
- Without **rules of syntax**, it would be **tedious** to understand programmer's intention
- Without clearly defined **instruction semantics**: difficult to write programs to solve specific problems & to build CPUs that do "right" thing (**Ambiguity In, Ambiguity Out!**)

C code:

```
int i;  
int sum = 0;  
  
for (i = 0; i < 10; i = i + 1)  
    sum = sum + i;
```

Decremented Loop

C code:

```
int i;  
int sum = 0;  
  
for (i = 10; i >= 1; i = i - 1)  
    sum = sum + i;
```

Assembly code:

```
; R0 = i  
; R1 = sum  
MOV R0, #9  
MOV R1, #0  
FOR  
    ADD R1, R1, R0  
    SUBS R0, R0, #1  
    BNE FOR  
DONE
```

- add sum to i
- i-- and set flags
- if i!=0 keep looping

- Saves 2 instructions per iteration compared to optimized (increment) version
 - Decrement loop variable & compare: SUBS R0, R0, #1
 - Only 1 branch – instead of 2
- MANY ways to solve (transform) a high-level problem into assembly
 - Code Optimization: A sub-field of Compilers
 - Aims to minimize total instruction count, branch instruction count, and maximize register utilization (to avoid frequent trips to memory)

For Loop

- Repeat **TEN** times: **add 10 to R1**
 - What is wrong with the code below (one way to think of a FOR loop)?
- Poor practice
- Code is not reusable
 - Next time it may be 20 not 10
- Instructions cost Memory!!
 - Each instruction is stored in memory and has an address
 - Memory is expensive!
 - Fast Instruction Cache built out of SRAM inside CPU is very premium
- How many instructions for above with a For loop using branch instruction?

```
ADD R1, R1, #10
```

While Loop

While Loop in C

- While loops **iterate** a number of times until the “controlling condition” is not met (**FALSE**)

```
C code:  
while (CONDITION) {  
    ...  
    ...  
}
```

- The following while loop executes forever!

```
C code:  
while (TRUE) {  
    ...  
    ...  
}
```

Example While Loop

- Determine X such that $2^X = 128$

C code:

```
int pow = 1;
int x = 0;

while (pow != 128) {
    pow = pow * 2;
    x = x + 1;
}
```

Assembly code:

```
; R0 = pow
; R1 = X
    MOV  R0, #1
    MOV  R1, #0
WHILE
    CMP  R0, #128
    BEQ  DONE
    LSL  R0, R0, #1
    ADD  R1, R1, #1
    B    WHILE
DONE
```

- loop initialization
- $\text{pow} = 1$
- $\text{X} = 0$
- $\text{pow} \neq 128?$
- if $\text{pow} == 128$, exit loop
- $\text{pow} = \text{pow} * 2$
- $\text{X} = \text{X} + 1$
- repeat loop

Arrays

Data Structure: Collection of data values organized in a particular way for ease of storage and access. Two aspects: organization and functions to access (i.e., read and write values)

Examples: Array, Linked List, Stack, Queue



Recall: Address vs. Value



- Square brackets signify **address** (also called **pointer** in C)

LDR R3, [R2, #8]

- If you [add the contents of register R2 (a binary number) to constant #8, you will get the **address** with which to **access** memory]
 ^ **Base + Offset Addressing Mode**
- When presented with an address, memory obliges by returning data stored at address
- In a 32-bit computer
 - Width of address bus = 32 bits (address space = 2^{32} locations)
 - Although memory is byte-addressable, it returns a 32-bit word to fill the entire register

REGISTER can hold memory address

[R1] = R1 is a pointer (\rightarrow) to Data

Memory Load returns Data or Value

Data is Stored in memory. Address is INPUT

Same Memory Stores Instructions and Data

[PC] \rightarrow Instruction

What is an Array?

- **Array:** A list of **data objects** of the same **type** arranged **sequentially in memory**

Array of 1-Byte Objects



Array of 4-Byte Objects



- A **data object** is a memory location whose content represent “some” value
 - Post office box can store letters, Amazon gifts, pamphlets (all these are pkgs. **types**)
 - How do we know ***interpret*** the **type** of what is stored in the box?
 - Either we know **what we placed there**, or we know **how to look up the type**
- The **interpretation** of the value in memory depends on its **type**
 - 8-Byte Unsigned Integers (**unsigned int**)
 - 4-Byte 2's Complement Integers (**int**)
 - A 12-Byte student record with **{uint student_Id, int grade}**



Array in Memory

- The array below has six **elements** and each **element** is a single **byte**
 - The **index** of the first **element** (**byte**) is 0, then 1, then 2,
- It's **base** (starting) address in memory is 0
 - The address of the first element is 0, second element is 1, last element is 5



↑ Base Address = Address of the first element

- Another array with six elements



↑ Base Address = Address of the first element

- Same starting address as the first array and same indexing scheme (0, 1, 2, ...)
- **Addresses of array elements in memory are different**
 - Second element is at an offset 4, last one at 20. Offsets are in bytes

Array Syntax in C

- Arrays contain a collection of similarly typed elements
- Elements are stored contiguously in memory

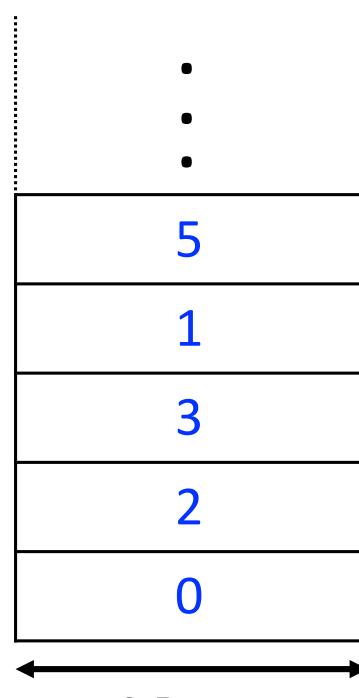
int is 4 bytes on most architectures

C code:

```
int marks[5] = {0, 2, 3, 1, 5};  
int a = marks[0];  
marks[3] = 10;
```

Address	Data	Index	Element
...
00000010	5	4	marks[4]
0000000C	1	3	marks[3]
00000008	3	2	marks[2]
00000004	2	1	marks[1]
00000000	0	0	marks[0]

4 Bytes



Array of Characters

- Array of **characters** (**char** is a data type in C)
- **char** is used for representing characters

char is always 1 byte	Address	Data	Index	Element
C code:
<code>char alphas[5] = {'a', 'b', 'c', 'd', 'e'};</code>	00000004	'e'	4	alphas[4]
	00000003	'd'	3	alphas[3]
	00000002	'c'	2	alphas[2]
	00000001	'b'	1	alphas[1]
	00000000	'a'	0	alphas[0]

↔ 1 Byte

Example Array in C

Add 10 to each element of the 200-element scores array

C code:

```
int i;
int scores[200];
// initialization code not
//shown
...
for (i = 0; i<200; i++)
    scores[i] = scores[i] + 10;
```

Array Sum

Add 10 to each element of the 200-element scores array

```
C code:  
int i;  
int scores[200];  
// initialization code not  
// shown  
...  
for (i = 0; i<200; i++)  
    scores[i] = scores[i] + 10;
```

address	data	
0x14000010	90	scores[4]
0x1400000C	76	...
0x14000008	80	scores[2]
0x14000004	40	scores[1]
base → 0x14000000	100	scores[0]

4 bytes

Showing the scores array in memory

Array Sum

Add 10 to each element of the 200-element scores array

C code:

```
int i;  
int scores[200];  
// initialization code not  
// shown  
...  
for (i = 0; i<200; i++)  
    scores[i] = scores[i] + 10;
```

address

90

76

80

40

100

scores[4]

...

scores[2]

scores[1]

scores[0]

base → 0x14000000

4 bytes

Assembly code:

```
; R0 = array base address  
; R1 = i  
MOV R0, #0x14000000  
MOV R1, 0  
LOOP  
    CMP R1, #200  
    BGE L3  
    LSL R2, R1, #2  
    LDR R3, [R0, R2]  
    ADD R3, R3, #10  
    STR R3, [R0, R2]  
    ADD R1, R1, #1  
    B LOOP  
L3
```

- R0 = base addr
- i = 0
- i < 200?
- no? exit loop
- word to byte
- R3 = scores[i]
- R3 = R3 + 10
- scores[i] += 10
- i = i + 1
- repeat

Showing the scores array in memory

LDR with Offset in Register

- New LDR variant

```
LDR R3, [R0, R2]  
      |dest|base|offset  
LDR Rd, [Rn, Rm]
```

- It is common to load from memory with **[base + offset] addressing mode**, where **offset** increments by “some” value during each loop iteration
- ISA provides **support** for such scenarios **to bridge the semantic gap b/w high-level code and assembly code**
 - ISA **evolution** eases the **software “burden”**
 - On the other hand, ISA implementation (i.e., microarchitecture) **becomes more involved** (recall the **RISC vs. CISC** debate)

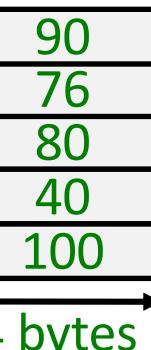
Array Sum

Add 10 to each element of the 200-element scores array

C code:

```
int i;
int scores[200];
// initialization code not
// shown
...
for (i = 0; i<200; i++)
    scores[i] = scores[i] + 10;
```

address



data
scores[4]
...
scores[2]
scores[1]
scores[0]

base → 0x14000000

Assembly code:

```
; R0 = array base address
; R1 = i
MOV R0, #0x14000000
MOV R1, #0
LOOP
    CMP R1, #200
    BGE L3
    LSL R2, R1, #2
    LDR R3, [R0, R2]
    ADD R3, R3, #10
    STR R3, [R0, R2]
    ADD R1, R1, #1
    B LOOP
L3
```

- R0 = base addr
- i = 0
- i < 200?
- no? exit loop
- word to byte
- R3 = scores[i]
- R3 = R3 + 10
- scores[i] += 10
- i = i + 1
- repeat

Showing the scores array in memory

Another LDR Variant

- We have seen two LDR variants
 - LDR Rd, [Rn, #imm]
 - LDR Rd, [Rn, Rm]
- LSL and LDR are often used together in array-related code (array traversals)
- ISA provides support for eliminating the extra LSL instruction

LDR R3, [R0, R1, LSL #2]



Left shift is the same
as multiplying by 2

- **Memory address**

- Left shift R1 by 2 (scaling R1)
- Add R1 to R0
- Address = R0 + (R1 * 4)

Condensing Array Sum – 1

Add 10 to each element of the 200-element scores array

C code:

```
int i;
int scores[200];
// initialization code not
// shown
...
for (i = 0; i<200; i++)
    scores[i] = scores[i] + 10;
```

address data

0x14000010	90	scores[4]
0x1400000C	76	...
0x14000008	80	scores[2]
0x14000004	40	scores[1]
base → 0x14000000	100	scores[0]

4 bytes

Assembly code:

```
; R0 = array base address
; R1 = i
MOV R0, #0x14000000
MOV R1, #0
LOOP
    CMP R1, #200
    BGE L3
    LDR R3, [R0, R1, LSL, #2]
    ADD R3, R3, #10
    STR R3, [R0, R2]
    ADD R1, R1, #1
    B LOOP
L3
```

Showing the scores array in memory

ARM Indexing Modes

- **Offset Addressing**
 - Address is the sum of base register and offset (#20, #-20, -R2)
 - Base register is unchanged
 - LDR R0, [R1, R2]
- **Pre-indexed Addressing**
 - Address is the sum of base register and offset
 - Base register is **updated** with the new address **before** the memory access
 - LDR R0, [R1, R2]!
- **Post-index Addressing**
 - Address is the base register
 - Base register is updated with the new address **after** the memory access
 - LDR R0, [R1], R2

Examples: ARM Indexing Modes

- **Offset Addressing**
 - LDR R0, [R1, R2]
 - **Address:** R1 + R2 and R1 does not change
- **Pre-indexed Addressing**
 - LDR R0, [R1, R2]!
 - **Address:** R1 + R2 and R1 = R1 + R2
- **Post-index Addressing**
 - LDR R0, [R1], R2
 - **Address:** R1 and R1 = R1 + R2
- **Note:** In all cases, offset can be an immediate

Condensing Array Sum – 2

Add 10 to each element of the 200-element scores array

C code:

```
int i;  
int scores[200];  
// initialization code not  
// shown  
...  
for (i = 0; i<200; i++)  
    scores[i] = scores[i] + 10;
```

address data

0x14000010	90	scores[4]
0x1400000C	76	...
0x14000008	80	scores[2]
0x14000004	40	scores[1]
base → 0x14000000	100	scores[0]

4 bytes

Assembly code:

```
; R0 = array base address  
; R1 = i  
MOV R0, #0x14000000  
MOV R1, R0, #800  
LOOP  
    CMP R0, R1  
    BGE L3  
    LDR R2, [R0]  
    ADD R2, R2, #10  
    STR R2, [R0], #4  
    B LOOP  
L3
```

- R0 = base addr
- R1 = base + 800
- end of array?
- yes? exit loop
- R2 = scores[i]
- scores[i] + 10
- store scores[i]
- and R0 = R0 + 4
- repeat loop

Showing the scores array in memory

Condensing Array Sum – 2

Add 10 to each element of the 200-element scores array

Assembly code:

```
; R0 = array base address
; R1 = i
    MOV  R0,  #0x14000000
    MOV  R1,  R0,  #800
LOOP
    CMP  R0,  R1
    BGE L3
    LDR  R2,  [R0]
    ADD  R2,  R2,  #10
    STR  R2,  [R0], #4
    B    LOOP
L3
```

- This version of Array Sum first computes the address of the last byte of the array (#0x14000800)
- Each iteration of LOOP checks if R0 is greater than or equal to #0x14000800
- If so, we are done, so step out of LOOP
- STR R2, [R0], #4
 - Stores R2 at [R0], and after that, adds 4 to R0

Addressing Modes

Addressing Modes

- Addressing mode **specifies** how instruction operands are **addressed**
 - Source and destination registers
 - Target address of a memory reference
 - Target address that a branch will jump to
- ARM uses four main modes
 - Register
 - Immediate
 - Base
 - PC-relative
- First three modes for reading/writing operands
 - Last mode is for writing the program counter

ARM Addressing Modes

- Some of the addressing modes allow the second source operand to be shifted
 - Check your references for details

Table 6.12 ARM operand addressing modes

Operand Addressing Mode	Example	Description
Register		
Register-only	ADD R3, R2, R1	$R3 \leftarrow R2 + R1$
Immediate-shifted register	SUB R4, R5, R9, LSR #2	$R4 \leftarrow R5 - (R9 \gg 2)$
Register-shifted register	ORR R0, R10, R2, ROR R7	$R0 \leftarrow R10 (R2 ROR R7)$
Immediate	SUB R3, R2, #25	$R3 \leftarrow R2 - 25$
Base		
Immediate offset	STR R6, [R11, #77]	$\text{mem}[R11+77] \leftarrow R6$
Register offset	LDR R12, [R1, -R5]	$R12 \leftarrow \text{mem}[R1 - R5]$
Immediate-shifted register offset	LDR R8, [R9, R2, LSL #2]	$R8 \leftarrow \text{mem}[R9 + (R2 \ll 2)]$
PC-Relative	B LABEL1	Branch to LABEL1

Addressing Mode Tradeoffs



- Complex addressing modes simplify high-level code to assembly translation
- But they result in more complex circuits (microarchitecture)
 - Shifter in front of ALU
 - ALU to add base and offset
- Where to place the burden of optimization?
 - Software or Hardware
 - Many simple instructions + Simple microarchitecture
 - Few complex instructions + Complex microarchitecture

ISA Tradeoffs



ISA Impacts Software and Hardware

- **Complex instructions**
 - (Upside) Dense and efficient code
 - (Downside) Complex circuits with longer critical paths
 - Example: x86 operate instructions can have both register and memory operands
 - Register-Memory architecture
- **Simple instructions**
 - (Upside) Simple circuits (microarchitecture)
 - (Downside) Large instruction footprint
 - (Downside) Big semantic gap between high-level code and assembly code
 - Example: ARM allows accessing memory only via LDR/STR
 - Load-Store architecture
- **Number of Registers**
 - Large register file demands more space in the ISA for encoding
 - Reduced trips to memory (memory references)

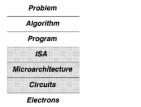
ISA Impacts Software and Hardware

- ISA impacts
 - Performance
 - Power and energy
 - Code size and instruction footprint
 - Circuit cost and complexity (chip area)
 - Future growth (ISA evolution)

Requirements, Bottlenecks, and Good Fortune: Agents for Microprocessor Evolution

YALE PATT, FELLOW, IEEE

Invited Paper



The first microprocessor, the Intel 4004, showed up in 1971. It contained about 2,300 transistors and ran at 100 kHz. Today, 30 years later, the microprocessor contains almost 200 million transistors and runs at 3 GHz. In five years, it will run at 10 GHz. In five years, these numbers are expected to grow to more than a billion transistors on a single chip, operating at a clock frequency of 40 GHz.

The evolution of the microprocessor, from where it started in 1971 to where it is today and what likely to be in 5 years, has come about through several causes and events. One reason is that this evolution did not just happen, that each step forward came as a result of a series of decisions made by computer architects and of a computer architect making tradeoffs. The three things are: 1) new requirements; 2) bottlenecks; and 3) good fortune. I call them collectively "agents."

This article attempts to do three things: describe a basic framework for understanding the evolution of the microprocessor, show some of the major agents that have influenced the evolution, discuss some developments that have come along in the 30 years since the arrival of the first microprocessor, and finally, suggest some of the new requirements, bottlenecks, and good fortune that will affect the evolution of the microprocessor in the next five years.

Keywords—Computer architecture, microarchitecture, microprocessor, microprocessor design, microprocessor evolution.

Fig. 1. The microprocessor today.

tradoffs. This has been especially true throughout the evolution of the microprocessor.

B. Levels of Transformation

Numbers of transistors, and their switching times are resources provided by process technology. What we use these resources for depends on the demands of the marketplace. How we use those resources is what the microprocessor is all about. Fig. 1 shows the levels of transformation in a problem-solving process. Just like English, most go through to be solved. In a real sense, it is the electrons that actually do the work and solve the problem. However, since we do not speak "electronics," we cannot do not speak any language. So, how best we can do is to systematically transform the problem through the levels shown on Fig. 1 until we reach the electrons (or device) level, that is, the 200 million transistors.

Along the way, the problem solution is first formulated as

100% to get we valuable characteristics of

change, h 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100

Next Time: Microarchitecture

Suggested Reading: Requirements, Bottlenecks, and Good Fortune:
Agents for Microprocessor Evolution

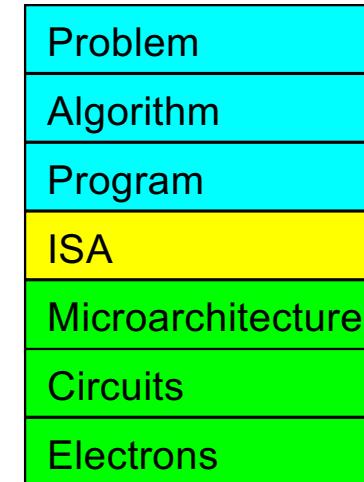
Link: https://course.ece.cmu.edu/~ece740/f13/lib/exe/fetch.php?media=r0_patt.pdf

Summary: ARM Inst. Formats

	31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
DP-I	cond	00	1	cmd	S	Rn	Rd	0 0 0 0	imm8
	31:28	27:26	25	24:21	20	19:16	15:12	11:4	3:0
DP-R	cond	00	0	cmd	S	Rn	Rd	0 0 0 0 0 0 0 0	Rm
	31:28	27:26	25:20		19:16	15:12		11:0	
Mem	cond	01	1 1 1 0 0 L		Rn	Rd		imm12	
	31:28	27:26	25:24				23:0		
BR	cond	10	10				imm24		

Today's Lecture

- Last few lectures
 - Instruction Set Architectures (**ISAs**): ARM and QuAC
 - Assembly programming: ARM
- **Today: Microarchitecture**
 - **Implementation of the ISA** (arrangement of registers, memories, ALU, other blocks)
 - Many different microarchitectures for one ISA are possible
 - **Design Point:** Set of considerations for a given problem space (ML, automotive)
 - **Leads to Tradeoffs:** Performance, power, reliability, cost, complexity
- **Today:** Design process and principles, single-cycle microarchitecture, and performance analysis
- Other **microarchitectures** we will cover
 - Multi-cycle, pipelined, and out-of-order



Many ISAs, Many Microarchitectures

- There can be many implementations of the same ISA
 - **MIPS** R2000, R3000, R4000, R6000, R8000, R10000, ...
 - **x86**: Intel 80486, Pentium, Pentium Pro, Pentium 4, Kaby Lake, Coffee Lake, Comet Lake, Ice Lake, Golden Cove, Sapphire Rapids, ..., AMD K5, K7, K9, Bulldozer, BobCat, Ryzen X, ...
 - **POWER** 4, 5, 6, 7, 8, 9, 10 (IBM), ..., **PowerPC** 604, 605, 620, ...
 - **ARM** Cortex-M*, ARM Cortex-A*, NVIDIA Denver, Apple A*, M1, ...
 - **Alpha** 21064, 21164, 21264, 21364, ...
 - **RISC-V** ...

How do we implement an ISA?

In other words, how do we design a system that obeys the hardware/software interface?

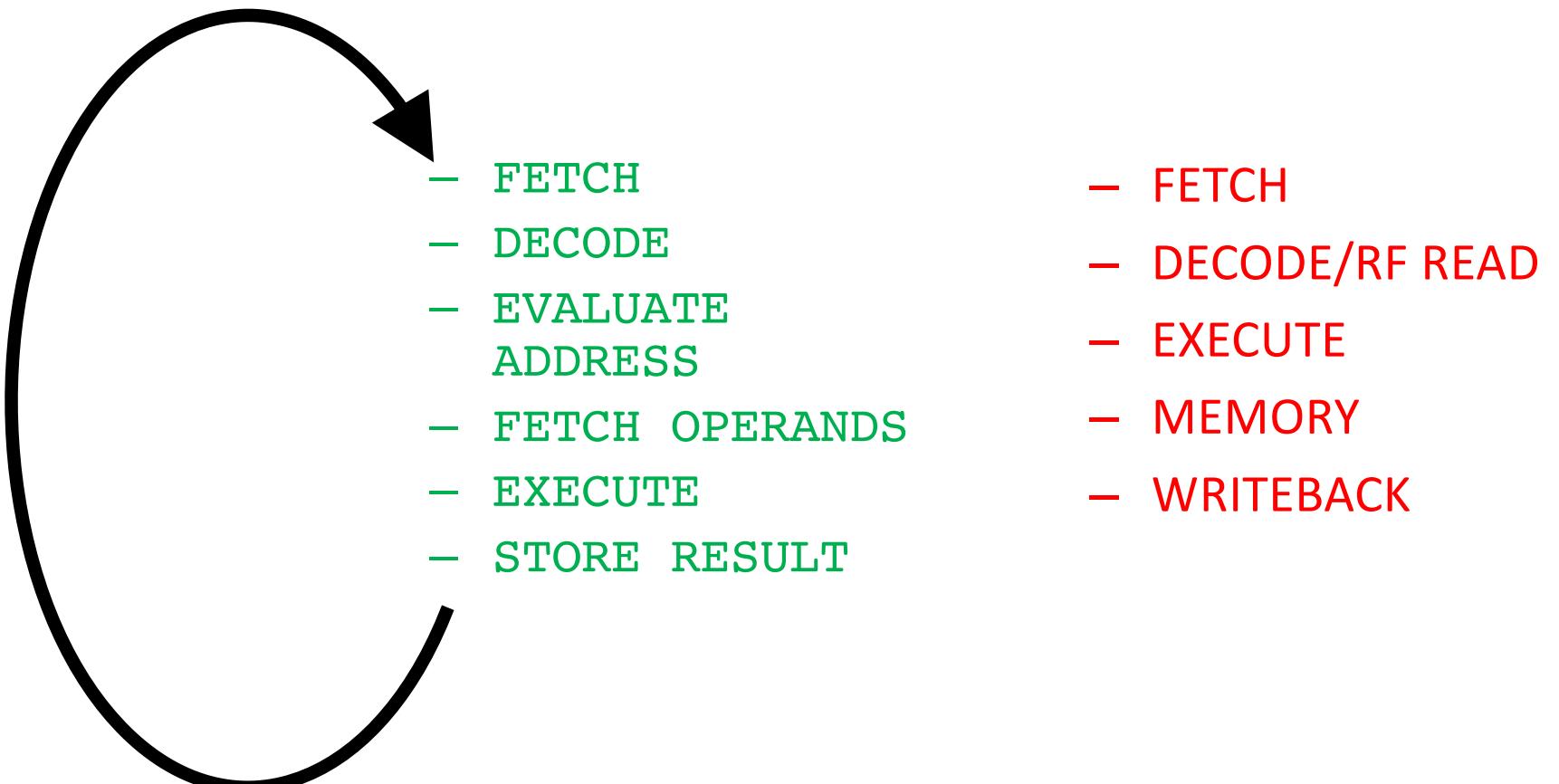
“Form follows function.”

Louis Sullivan



Before we begin construction, let's pause and ask: what is the purpose of this computer?

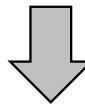
Purpose: To Process Instructions



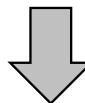
How does a machine process insts?

- What does processing an instruction mean in von Neumann model?
- We will assume the von Neumann model (for now)

AS = Architectural (programmer visible) state before an instruction is processed



Process Instruction



AS' = Architectural (programmer visible) state after an instruction is processed

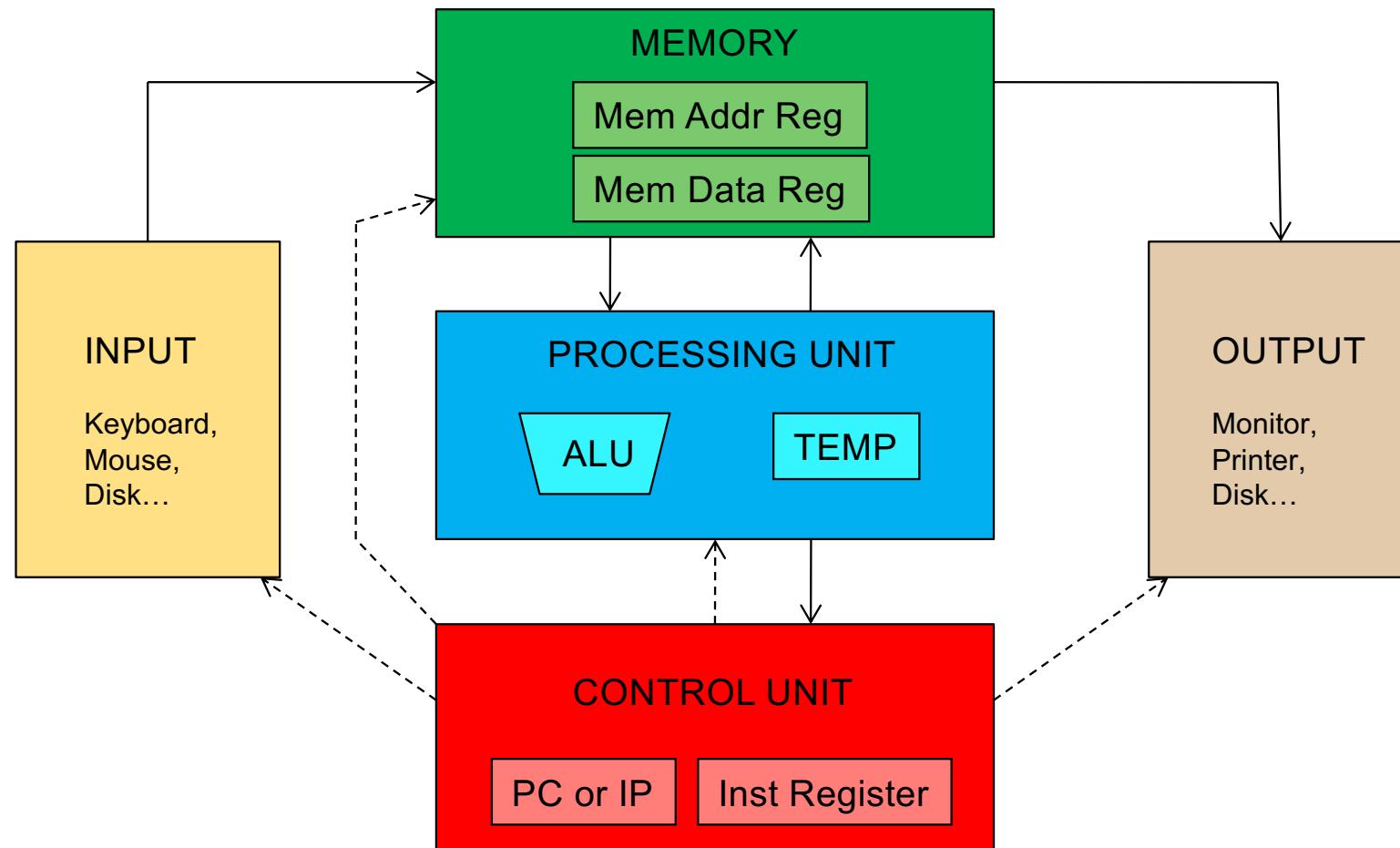
- Processing an instruction: Transforming AS to AS' according to the ISA specification of the instruction

The Von Neumann Model/Architecture

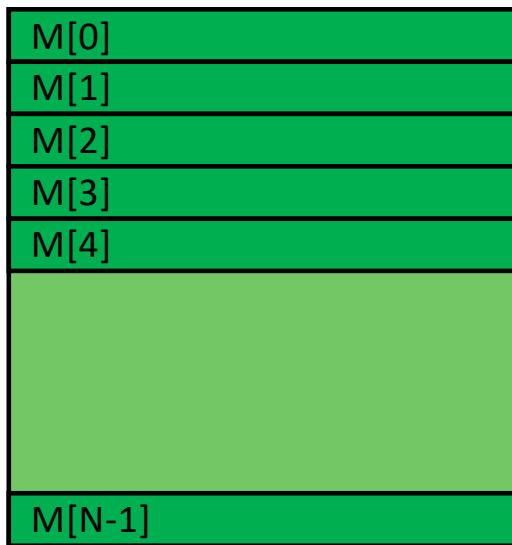
Stored program

Sequential instruction processing

The Von Neumann Model/Architecture

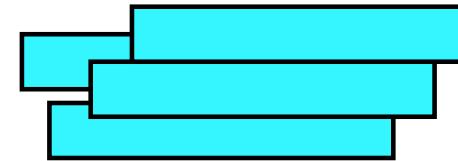


Recall: Programmer Visible (Architectural) State



Memory

array of storage locations
indexed by an address



Registers

- given special names in the ISA (as opposed to addresses)
- general vs. special purpose

Program Counter

memory address
of the current (or next) instruction

Instructions (and programs) specify how to transform
the values of programmer visible state

The “Process Instruction” Step

- ISA specifies abstractly what AS' should be, given an instruction and AS
 - It defines an **abstract finite state machine** where
 - State = programmer-visible state
 - Next-state logic = instruction execution specification
 - From ISA point of view, there are no “intermediate states” between AS and AS' during instruction execution
 - One state transition per instruction
- Microarchitecture implements how AS is transformed to AS'
 - There are many choices in implementation
 - We can have programmer-invisible state to optimize the speed of instruction execution: **multiple** state transitions per instruction
 - Choice 1: **AS → AS'** (transform AS to AS' in a single clock cycle)
 - Choice 2: **AS → AS+MS1 → AS+MS2 → AS+MS3 → AS'** (take multiple clock cycles to transform AS to AS')

Aside: What if a machine processes instructions out of program order?

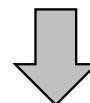
- What does the programmer care about?
- Does the programmer care if i3 executed before i4?
 - No: Programmer only cares R1 was updated before R0
 - Can update AS in program order and process instructions out of order (OOO)
- Why would a machine ever do that?
 - Fact: Almost EVERY high-performance computer does that!
 - In-program-order instruction processing (execution) is an illusion

```
i1: CMP R0, #10
i2: BGE DONE
i3: ADD R1, R1, R0
i4: ADD R0, R0, #1
```

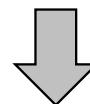
Basic Instruction Processing Engine

- Each instruction takes a single clock cycle to execute
- Only combinational logic is used to implement instruction execution
 - *No intermediate, programmer-invisible state updates*

AS = Architectural (programmer visible) state at the start of a clock cycle



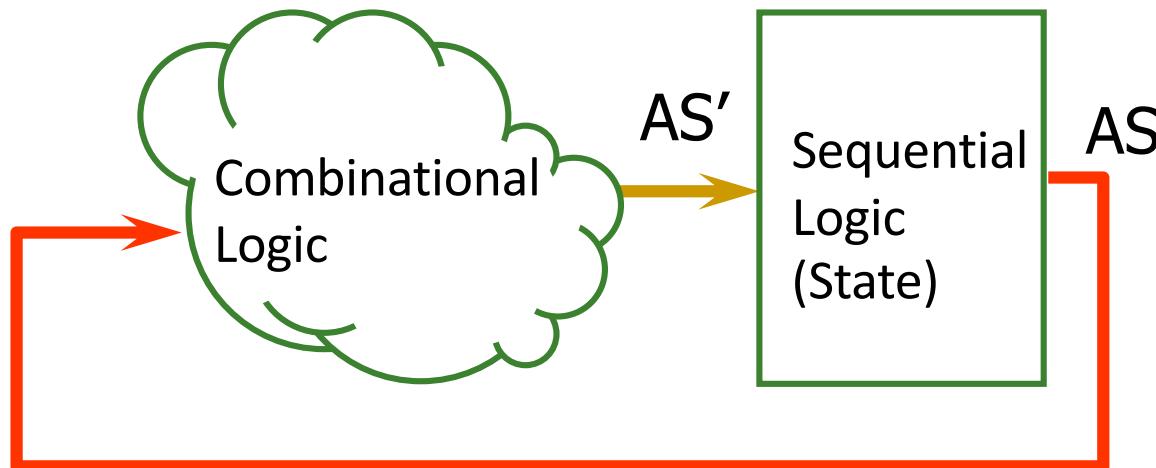
Process Instruction



AS' = Architectural (programmer visible) state at the end of a clock cycle

Basic Instruction Processing Engine

- Single-cycle machine



- What is the *clock cycle time* determined by?
- What is the *critical path* (i.e., longest delay path) of the combinational logic determined by?

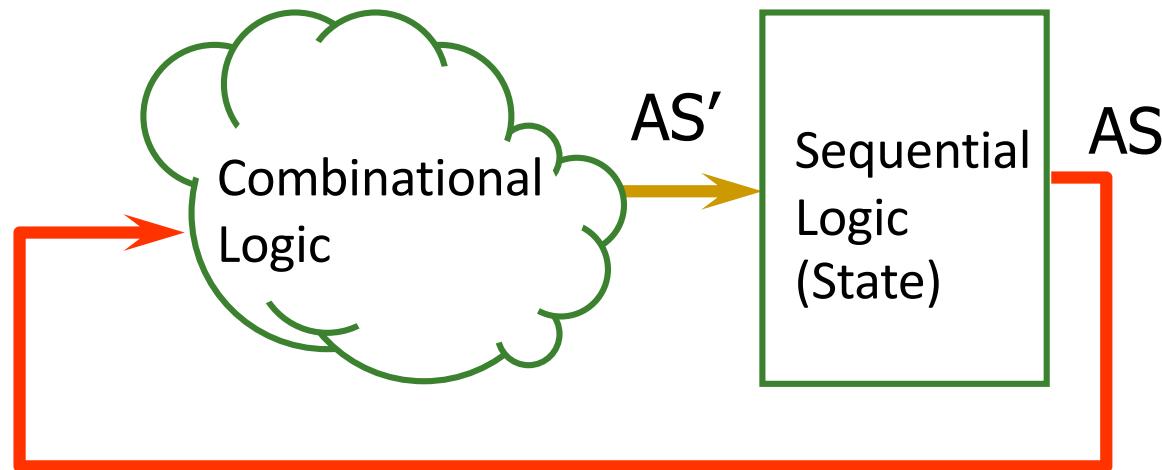
AS: Architectural State

Single-Cycle vs. Multi-Cycle Machines

- Single-cycle machines
 - Each instruction takes a single clock cycle
 - All state updates made at the end of an instruction's execution
 - Big disadvantage: The slowest instruction determines cycle time → long clock cycle time
- Multi-cycle machines
 - Instruction processing broken into multiple cycles/stages
 - State updates can be made during an instruction's execution
 - Architectural state updates made at the end of an instruction's execution
 - Advantage over single-cycle: The slowest “stage” determines cycle time
- Both single-cycle and multi-cycle machines literally follow the von Neumann model at the microarchitecture level

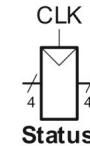
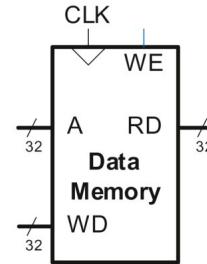
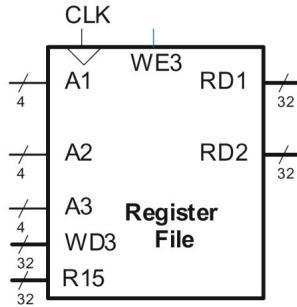
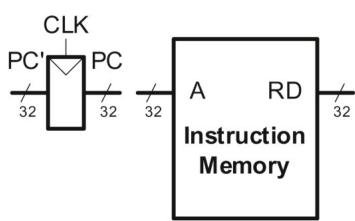
Remember

- Single-cycle machine



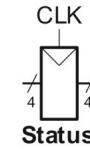
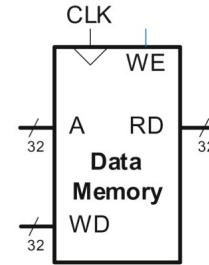
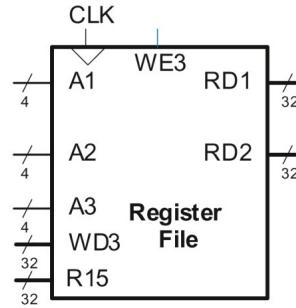
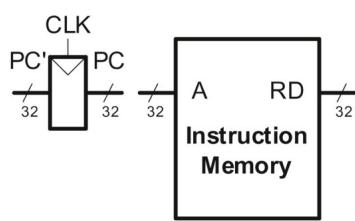
AS: Architectural State

ARM State (AS) Elements



- **PC:** Logically part of the register file
 - Read and written every cycle, independently of the normal register file operation. **Should it be part of the register file?**
- **Instruction memory** has a single read port. One 32-bit address input. One 32-bit instruction (RD) output.
- **Register file:** 15 registers (**R0** to **R14**) + additional input to receive **R15** from PC
 - Two 4-bit read ports A1 and A2 (32-bit RD1 and RD2)
 - One write port A3 (and WD3) and a **write enable** input
 - Read of R15 returns PC + 8
 - Write of R15 must be handled properly if PC is outside the register file
 - Reads are combinational and writes happen on the rising edge of the clock

ARM State (AS) Elements



- **Data Memory:** Single read/write port.
 - If write enable (WE) is TRUE then it writes data WD into address A on the rising edge of the clock.
 - If the write enable is FALSE, then it reads address A onto RD
- **Simplification**
 - All reads are combinational and constant time (not realistic but Ok for now)
- **All writes and state updates happen on the rising edge of the clock**
 - Synchronous sequential circuit

Microarchitecture Division

- Two interacting parts
 - Datapath (32-bit in our case)
 - Control unit
- Datapath **operates** on words of data
 - Memories, registers, ALUs, and multiplexers
- **Control** unit **informs** the datapath how to execute an instruction
 - Receives the current instruction from the datapath
 - Produces **multiplexer selects**, **ALU control**, **register enable**, and **memory write signals** to **control** the operation of the datapath

Microarchitecture Division

- Two interacting parts
 - Datapath (32-bit in our case)
 - Control unit
- Datapath **operates** on words of data
 - Memories, registers, ALUs, and multiplexers
- **Control** unit **informs** the datapath how to execute an instruction
 - Receives the current instruction from the datapath
 - Produces **multiplexer selects**, **ALU control**, **register enable**, and **memory write signals** to **control** the operation of the datapath

Role of Control Unit

Codes stored in memory control the hardware of the computer ... As a puppeteer controlling a troupe of marionettes in an acquisitively choreographed dance of arithmetic and logic. The CPU control signals are the strings.

CODE, Charles Petzold



Design Process

- We will add the logic for one instruction at a time
 - LDR (**LoaD Register**)
 - STR (**STore Register**)
 - Data Processing (**DP**) instructions with 2nd source operand as an immediate
 - DP with 2nd source operand as a register
 - Branch instruction
 - Control Unit

Design Process

- We limit ourselves to a subset of instructions
 - Data-processing instructions: ADD, SUB, AND, ORR (with register and immediate offsets but not shifts)
 - Memory instructions: LDR, STR (with positive immediate offset)
 - Branches: B
- Once you understand these you can expand the hardware to handle others

Design Process

- New connections are emphasized in black
- Hardware already studied in gray
- Control signals in blue

LDR with Src2 as Immediate

- I (Bit 25) = 1: Src2 = **imm12** where **imm12** is a **12-bit unsigned offset** added to the value in the base register (**Rn**)
- Format of **LoaD Register** instruction

LDR R0, [R1, #12]

↓ ↓ ↓

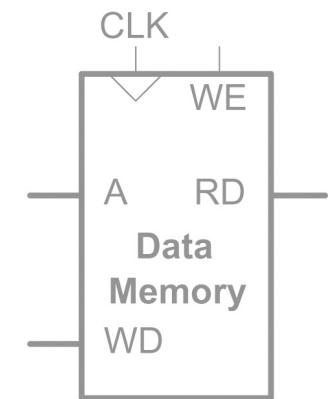
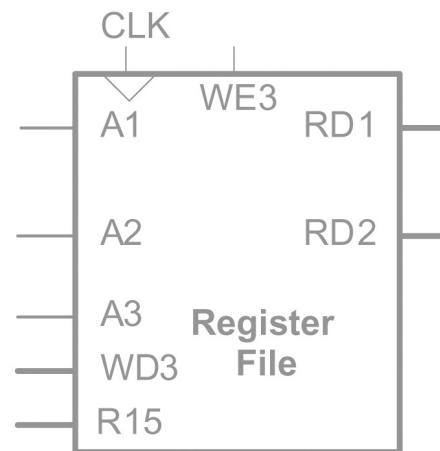
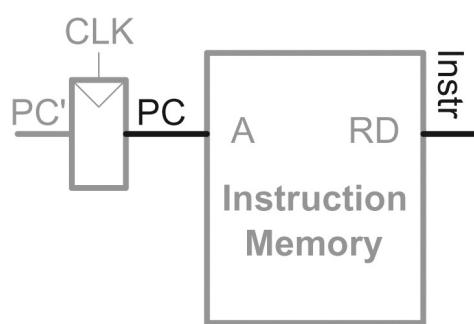
LDR Rd, [Rn, #imm12]

- L (Bit 20) = 1: CPU performs an **LDR**

	31:28	27:26	25:20	19:16	15:12	11:0
cond	01	1	1 1 0 0 1	Rn	Rd	imm12

The LDR Datapath

Step 1: Read (Fetch) instruction from memory

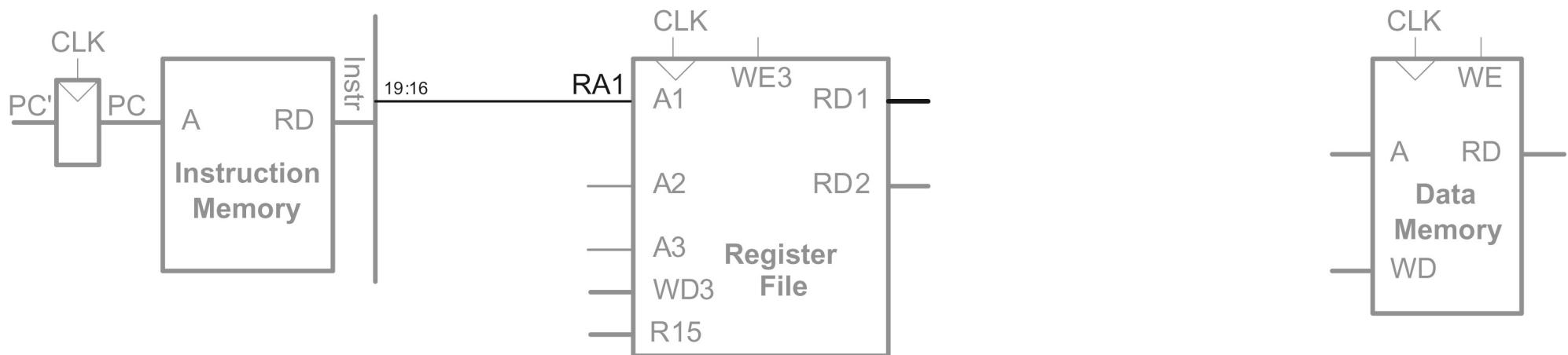


- Remember the distinction between PC and PC'
- From this point on, CPU actions depend on the instruction fetched

31:28	27:26	25:20	19:16	15:12	11:0
cond	01	1 1 1 0 0 L	Rn	Rd	imm12

The LDR Datapath

Step 2: Read source operand (base register, Rn) from register file

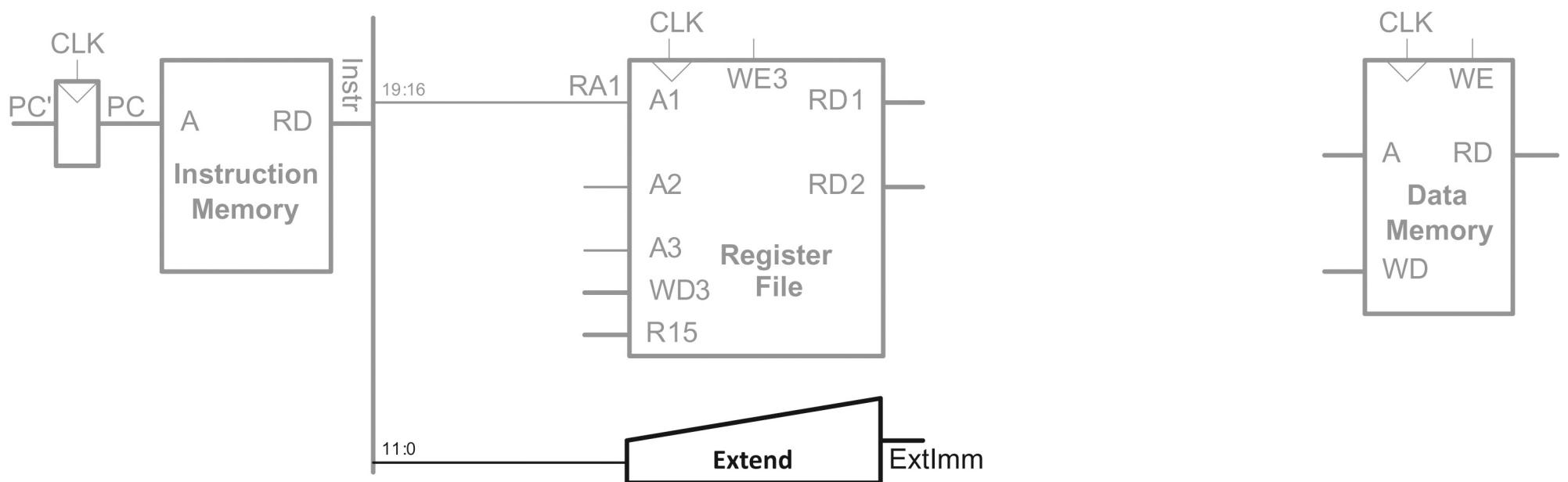


- Data is read onto RD1

31:28	27:26	25:20	19:16	15:12	11:0
cond	01	1 1 1 0 0 L	Rn	Rd	imm12

The LDR Datapath

Step 3: Zero-extend the immediate field stored in $\text{Instr}_{11:0}$



31:28	27:26	25:20	19:16	15:12	11:0
cond	01	1 1 1 0 0 L	Rn	Rd	imm12

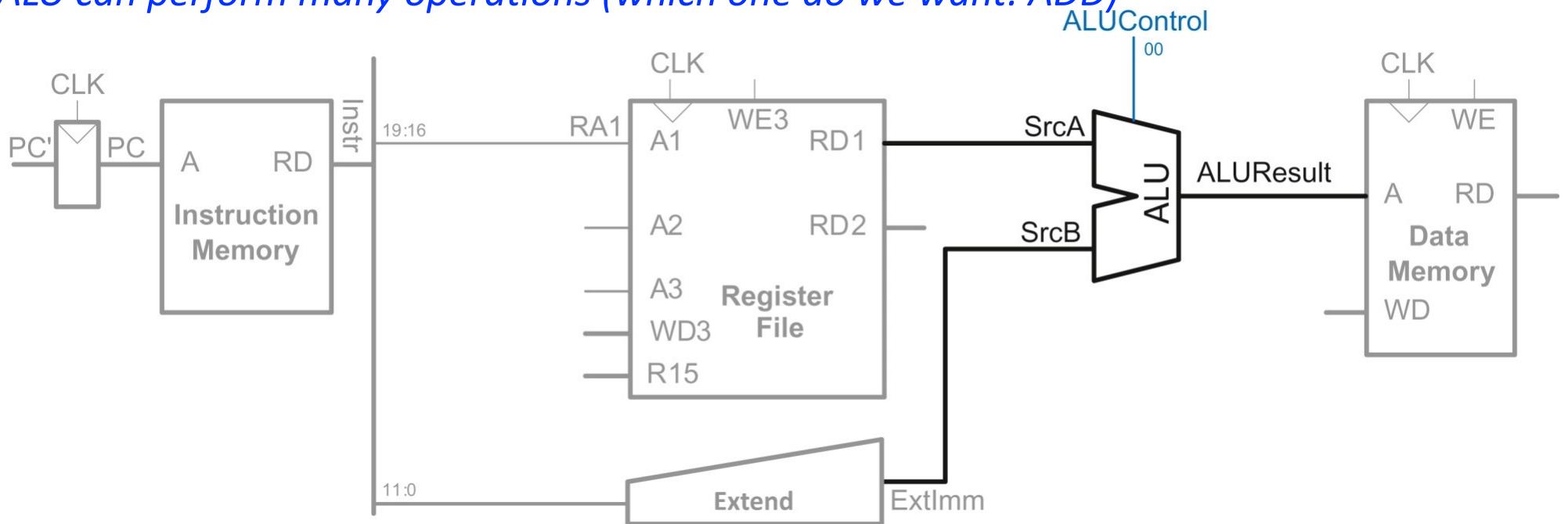
Zero Extension

- Appending leading zeros to make a smaller quantity equal to a bigger quantity
- $\text{ImmExt}_{31:12} = 0$ and $\text{ImmExt}_{11:0} = \text{Instr}_{11:0}$

The LDR Datapath

Step 4: Compute memory address (ALUControl = 00)

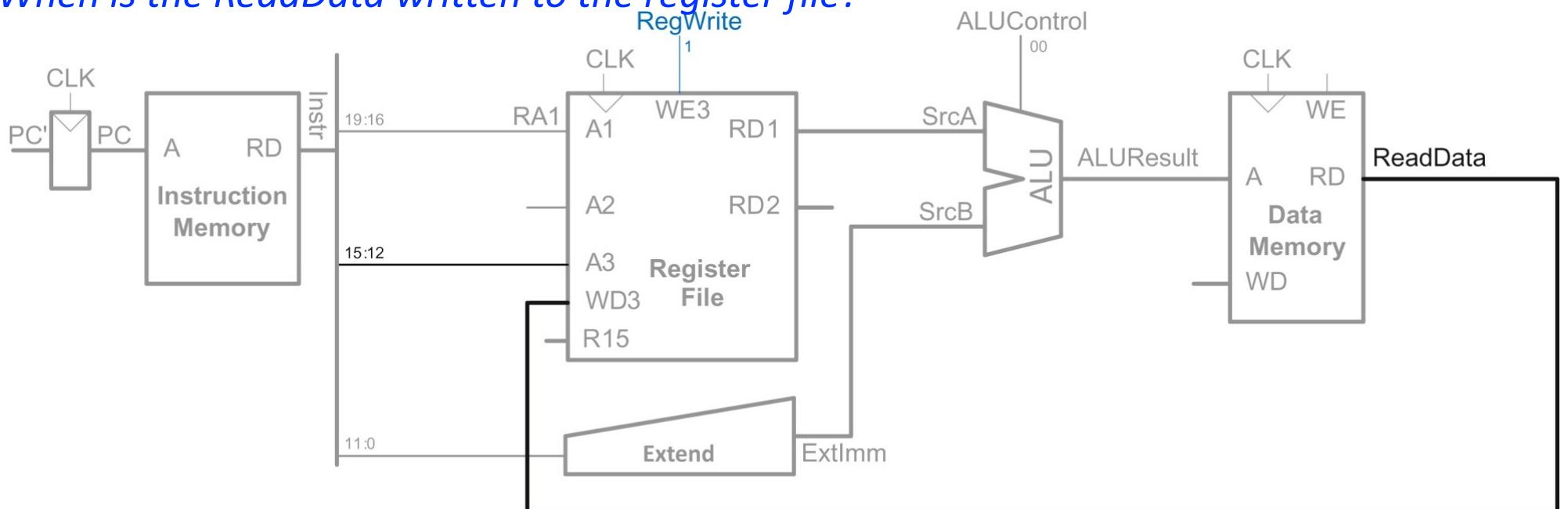
ALU can perform many operations (which one do we want: ADD)



31:28	27:26	25:20	19:16	15:12	11:0
cond	01	1 1 1 0 0 L	Rn	Rd	imm12

The LDR Datapath

Step 5: Write back data from read by data memory to Rd in Reg File
When is the ReadData written to the register file?



31:28

27:26

25:20

19:16

15:12

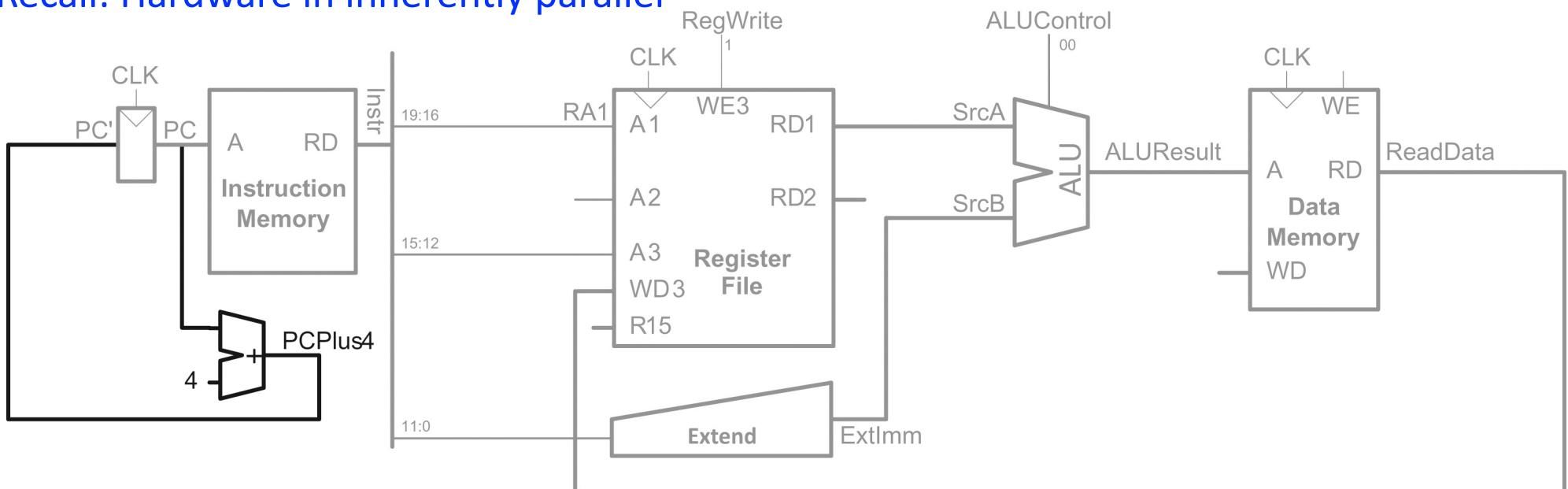
11:0

cond	01	1	1	1	0	0	L	Rn	Rd	imm12
------	----	---	---	---	---	---	---	----	----	-------

The LDR Datapath

Step 6: Compute address of next instruction ($PC' = PC + 4$)

Recall: Hardware is inherently parallel



PC will become PC' the following cycle (recall photography example)

31:28

27:26

25:20

19:16

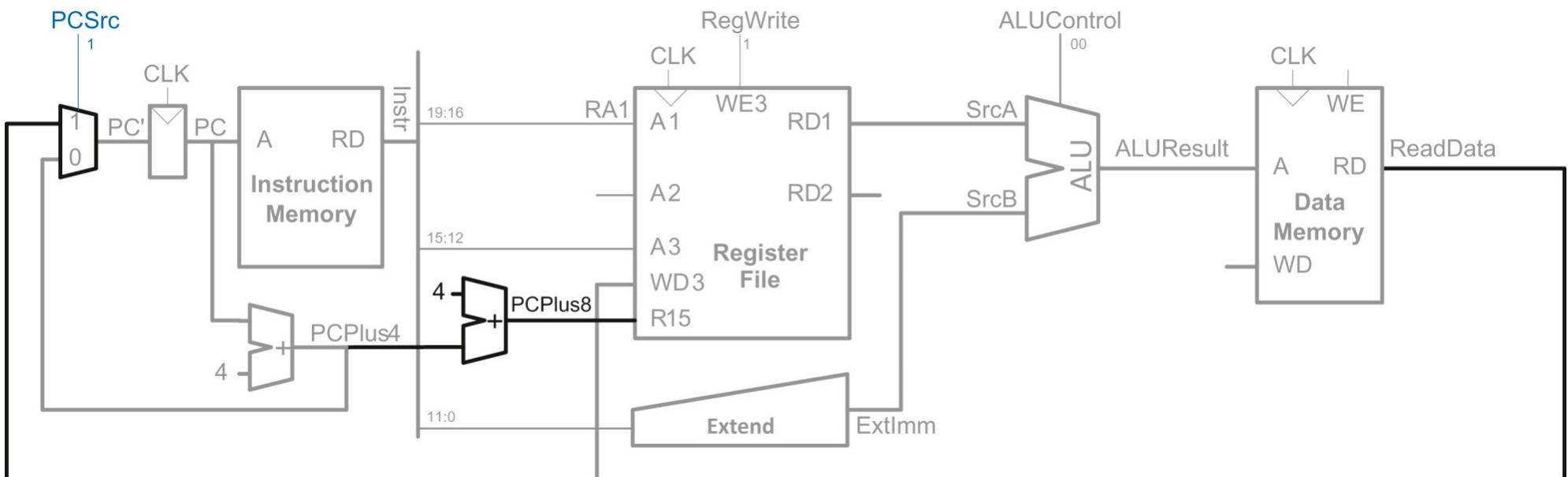
15:12

11:0

cond	01	1	1	1	0	0	L	Rn	Rd	imm12
------	----	---	---	---	---	---	---	----	----	-------

The LDR Datapath

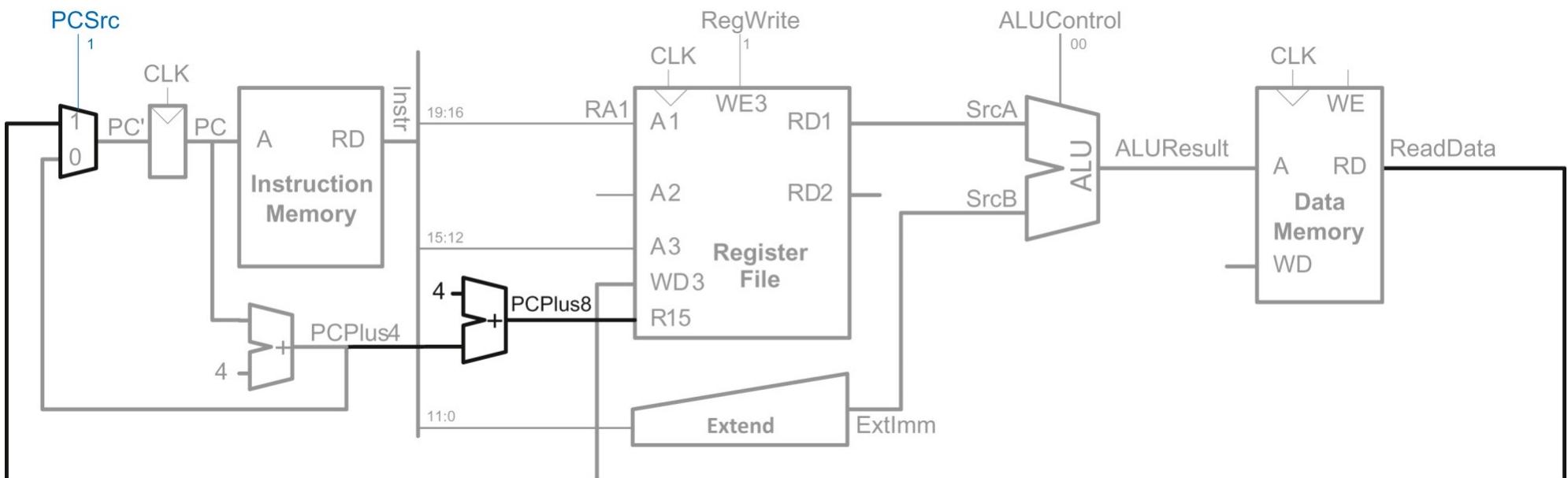
Step 7/a: Reading register R15 returns PC + 8



31:28	27:26	25:20	19:16	15:12	11:0
cond	01	1 1 1 0 0 L	Rn	Rd	imm12

The LDR Datapath

Step 7/b: Writing register R15 (PC may be an instruction's result)



31:28

27:26

25:20

19:16

15:12

11:0

cond	01	1	1	1	0	0	L	Rn	Rd	imm12
------	----	---	---	---	---	---	---	----	----	-------

STR Instruction

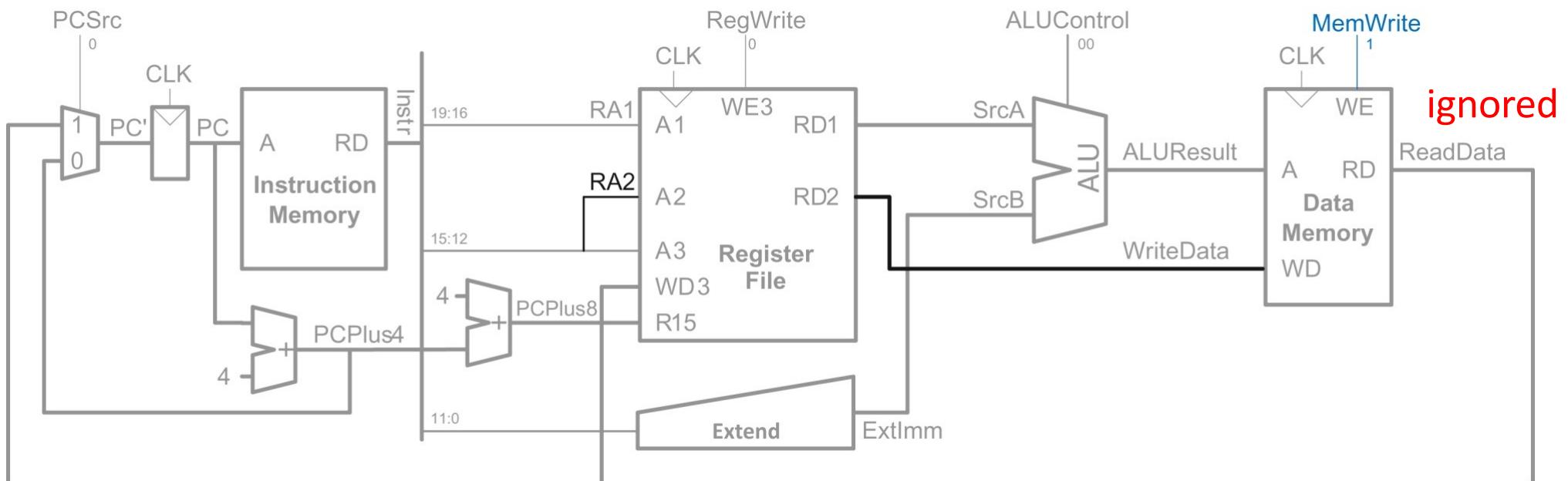
- STR instruction uses the same instruction format
 - LDR and STR behave differently at the machine level
- Rd is a source operand (specifies the register to store to mem)
- Format of STore Register instruction

STR R0, [R1, #12]
↓ ↓ ↓
STR Rd, [Rn, #imm12]

31:28	27:26	25:20	19:16	15:12	11:0
cond	01	1 1 1 0 0 L	Rn	Rd	imm12

The STR Datapath

Step 8: Read a second register (Rd) and write its value to memory



- **ReadData is ignored because RegWrite is FALSE**

31:28

27:26

25:20

19:16

15:12

11:0

cond	01	1	1	1	0	0	L	Rn	Rd	imm12
------	----	---	---	---	---	---	---	----	----	-------

DP Instructions: Immediate

- Like the LDR instruction, but two important differences
 - imm8 instead of imm12
 - The destination register stores the result of the ALU operation instead of memory access
- Format

ADD R0, R1, #16
ADD Rd, Rn, #imm8

31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0	
cond	00	1	cmd	S	Rn	Rd	0 0 0 0	imm8	

Adding Support for DP Instructions

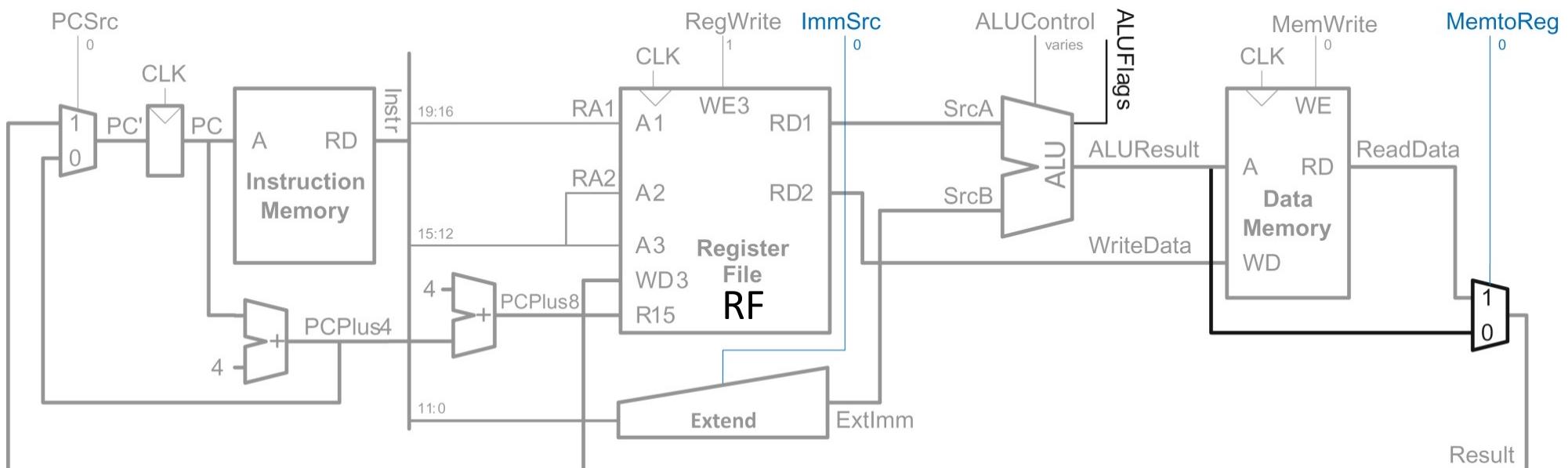
- The ALU functions and encoding

ALUControl	Function
00	ADD
01	SUB
10	AND
11	ORR

- The ALU also produces four flags that are sent to the control unit
- *Register file either receives data from the data memory or the ALU*
 - *Add a multiplexer to choose between ReadData and ALUResult*
 - *This multiplexer is controlled by MemtoReg*
 - *MemtoReg = 1 for LDR and 0 for data processing instructions*

DP-Immediate Datapath

Step 9: Change extend block, and add signal to write ALU result to RF



31:28 27:26 25 24:21 20 19:16 15:12 11:8 7:0

cond	00	1	cmd	S	Rn	Rd	0	0	0	0	imm8
------	----	---	-----	---	----	----	---	---	---	---	------

DP Instructions: Register

- The second source operand is Rm instead of an immediate
- Place Rm on the A2 port of the register file for DP instructions with register as the second operand
- Format

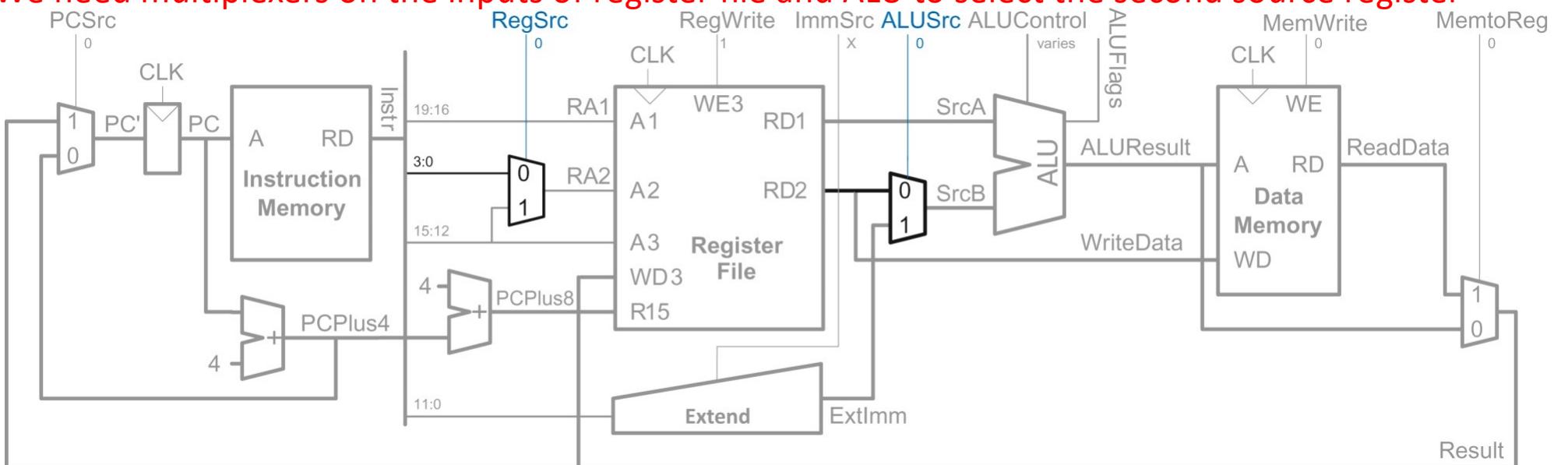
ADD R0, R1, R3
ADD Rd, Rn, Rm

31:28	27:26	25	24:21	20	19:16	15:12	11:4	3:0
cond	00	0	cmd	S	Rn	Rd	0 0 0 0 0 0 0 0	Rm

DP-Register Datapath

Step 10: Read 2nd register (Rm) from Reg File and send RD2 to ALU

We need multiplexers on the inputs of register file and ALU to select the second source register

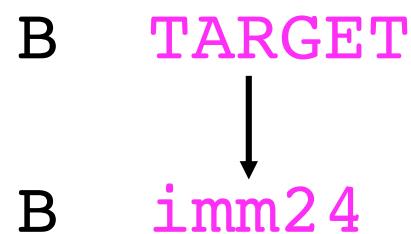


31:28 27:26 25 24:21 20 19:16 15:12 11:4 3:0

cond	00	0	cmd	S	Rn	Rd	0	0	0	0	0	0	Rm
------	----	---	-----	---	----	----	---	---	---	---	---	---	----

Branch Instruction: Unconditional

- The second source operand is Rm instead of an immediate
 - Place Rm on the A2 port of the register file for DP instructions with register as the second operand
 - Format



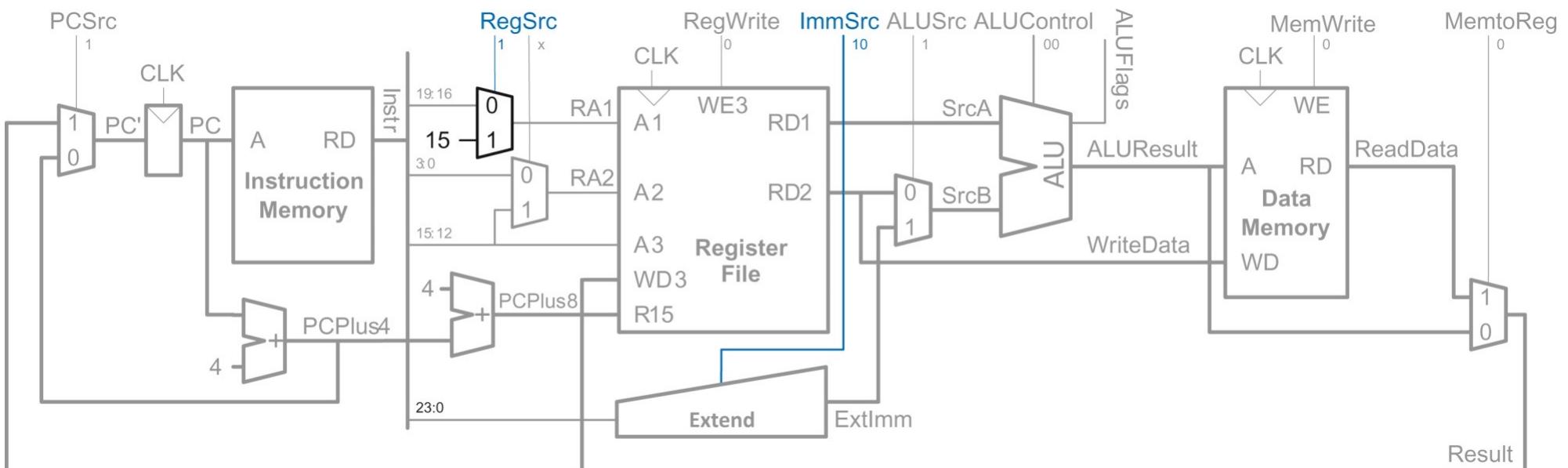
31:28 27:26 25:24

23:0

1110 | 10 | 10 | imm24

Branch Datapath

Step 11: Change extend block, and add a bit to RegSrc for branch



31:28 27:26 25:24

23:0

1110	10	10	imm24
------	----	----	-------

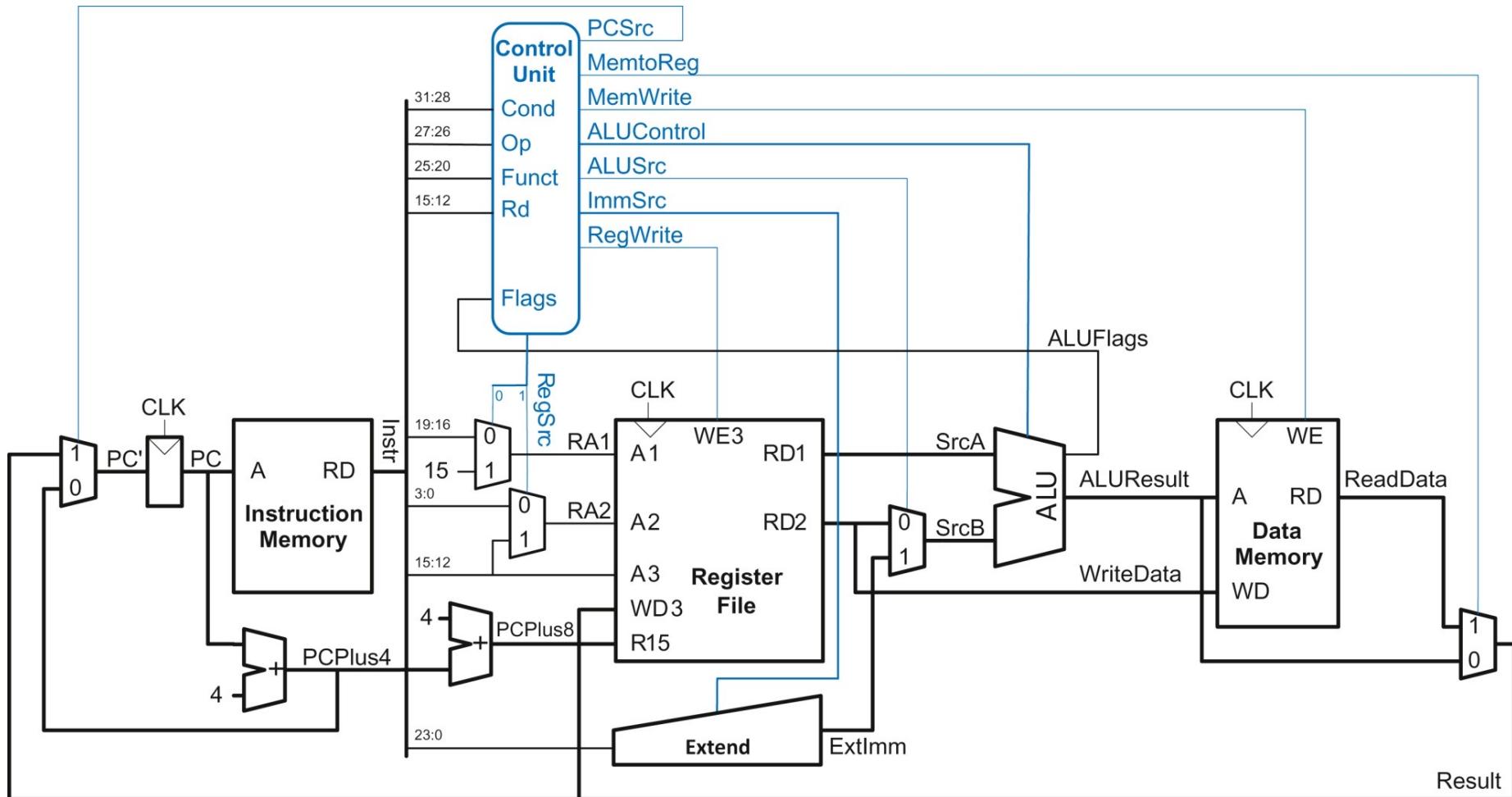
Operation of the Extend Block

Each of the three instruction formats interpret the immediate field differently

- $\text{ImmSrc}_{1:0}$ is the 2-bit control signal input to the extend block

ImmSrc_{1:0}	ExtImm	Description
00	{24'b0, Instr _{7:0} }	Zero-extended <i>imm8</i>
01	{20'b0, Instr _{11:0} }	Zero-extended <i>imm12</i>
10	{6{Instr ₂₃ }, Instr _{23:0} }	Sign-extended <i>imm24</i>

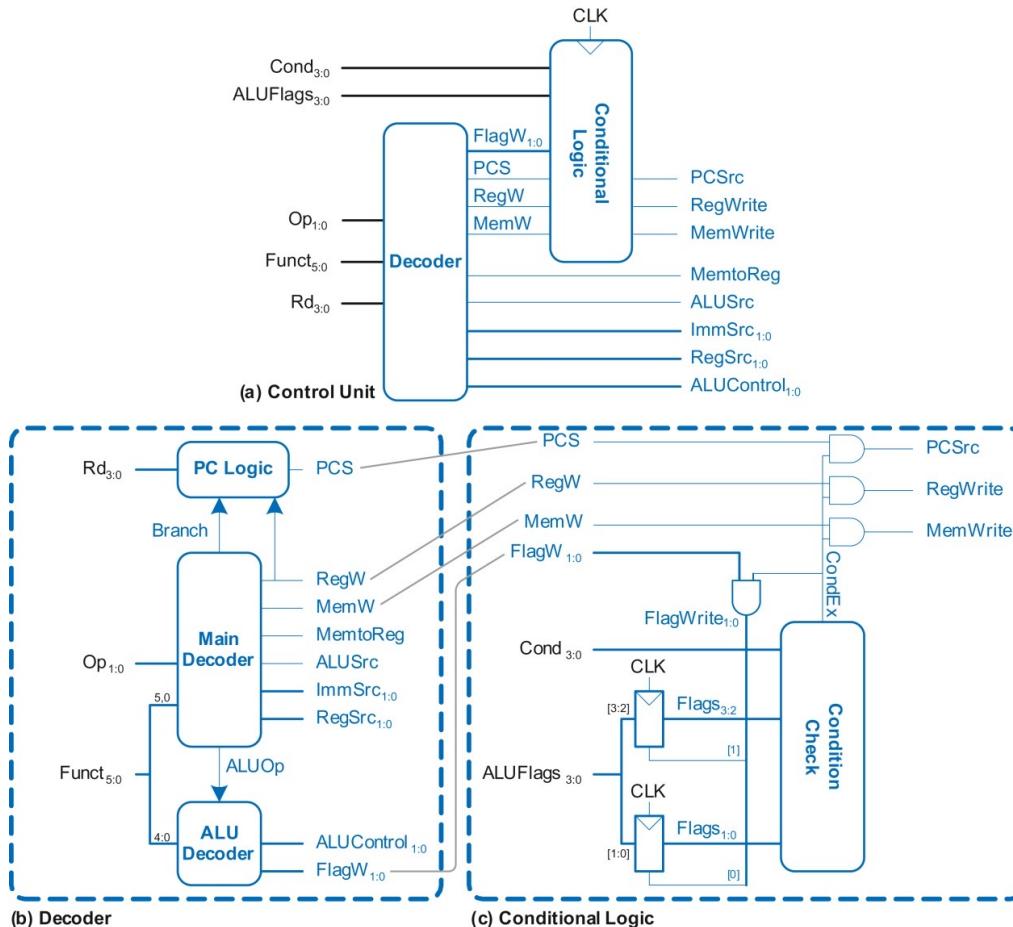
Datapath with Control



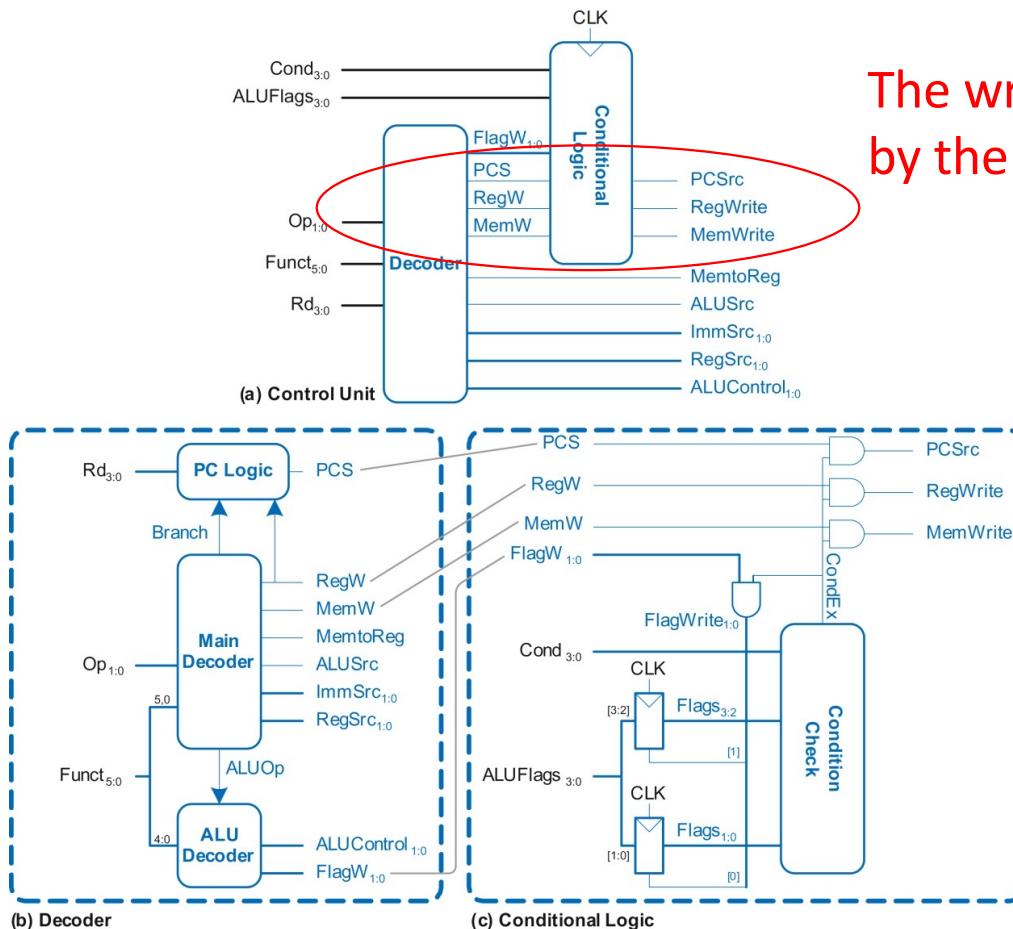
Control Unit

- Generate control signals based on instruction fields
 - $\text{Instr}_{31:20}$ (cond)
 - $\text{Instr}_{27:26}$ (op)
 - $\text{Instr}_{25:20}$ (funct)
 - Flags
 - Destination register (**why does the controller needs this?**)
- Controller for single-cycle microarchitecture is purely combinational
- Conditional logic must enable updates to the architectural state when the instruction should be conditionally executed
 - **Write enables must be TRUE only if conditional instruction is in fact executed**

One way to build the control unit



One way to build the control unit



Decoder Truth Table

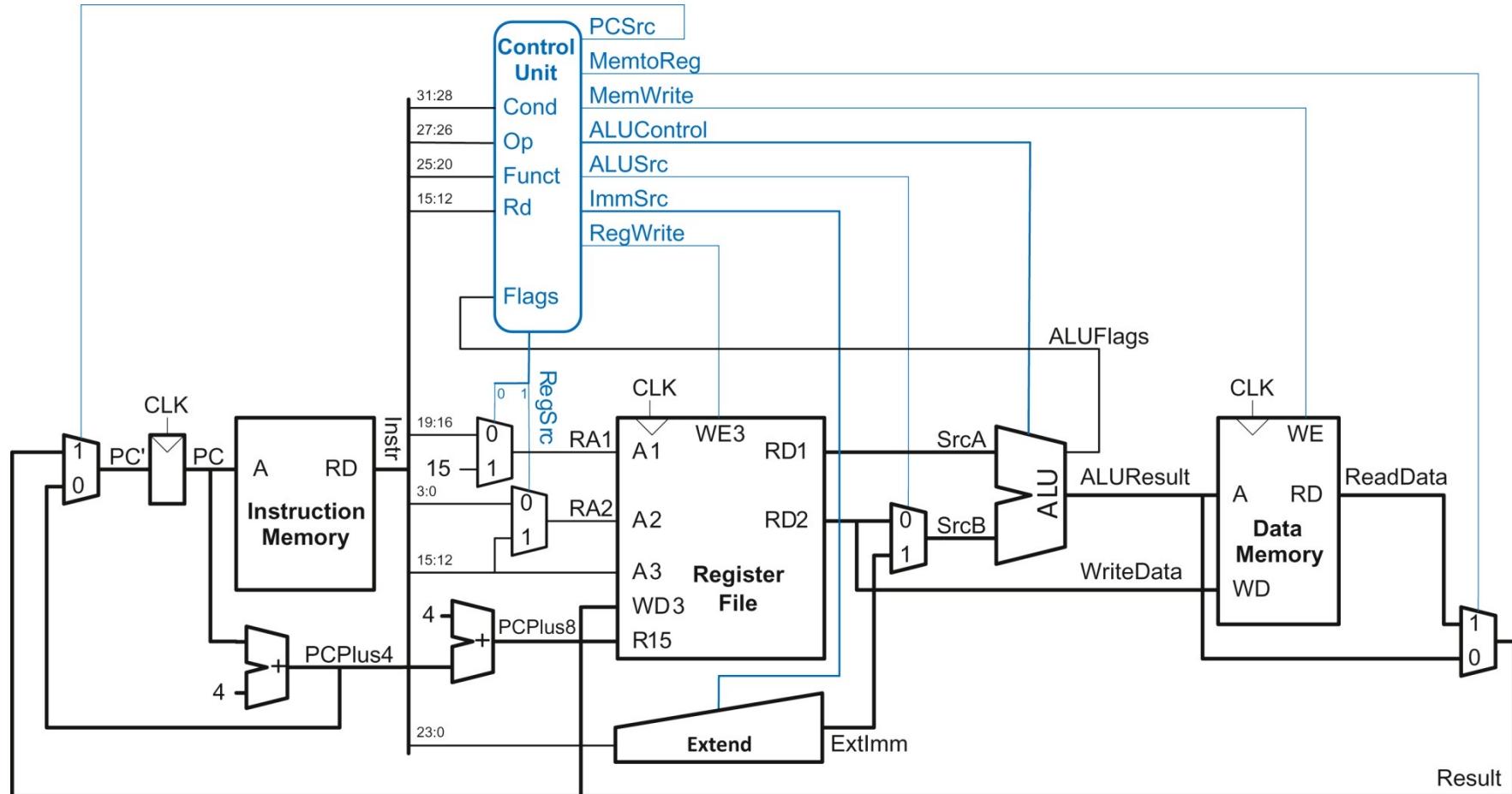
- Only selected signals are shown in the truth table

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
11	X	X	B	1	0	0	1	10	0	X1	0

Example: Generating PCSrc Signal

- PCSrc is **1** when
 - Destination register (Rd) is R15
 - RegW is **1** (ADD/SUB or LDR)
 - Instruction is a branch
- PCSrc = ((Rd == 15) & RegW) | Branch
 - Assuming the control unit generates a signal called Branch when opcode is 10 (B or BL)
- **Important:** Be careful to take conditional execution into account in the lab task and assignment!

Processor Operation: ORR

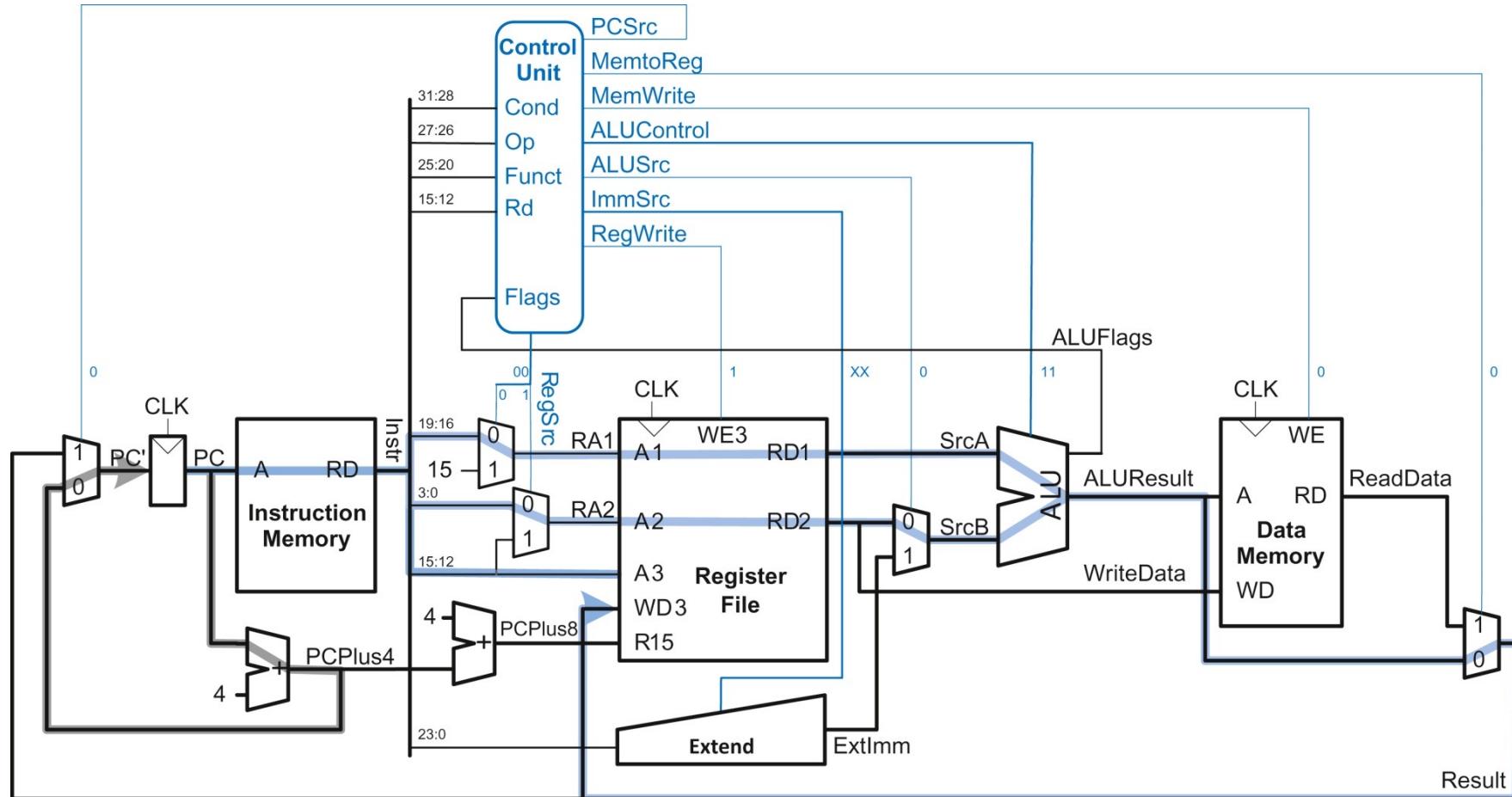


Processor Operation: ORR

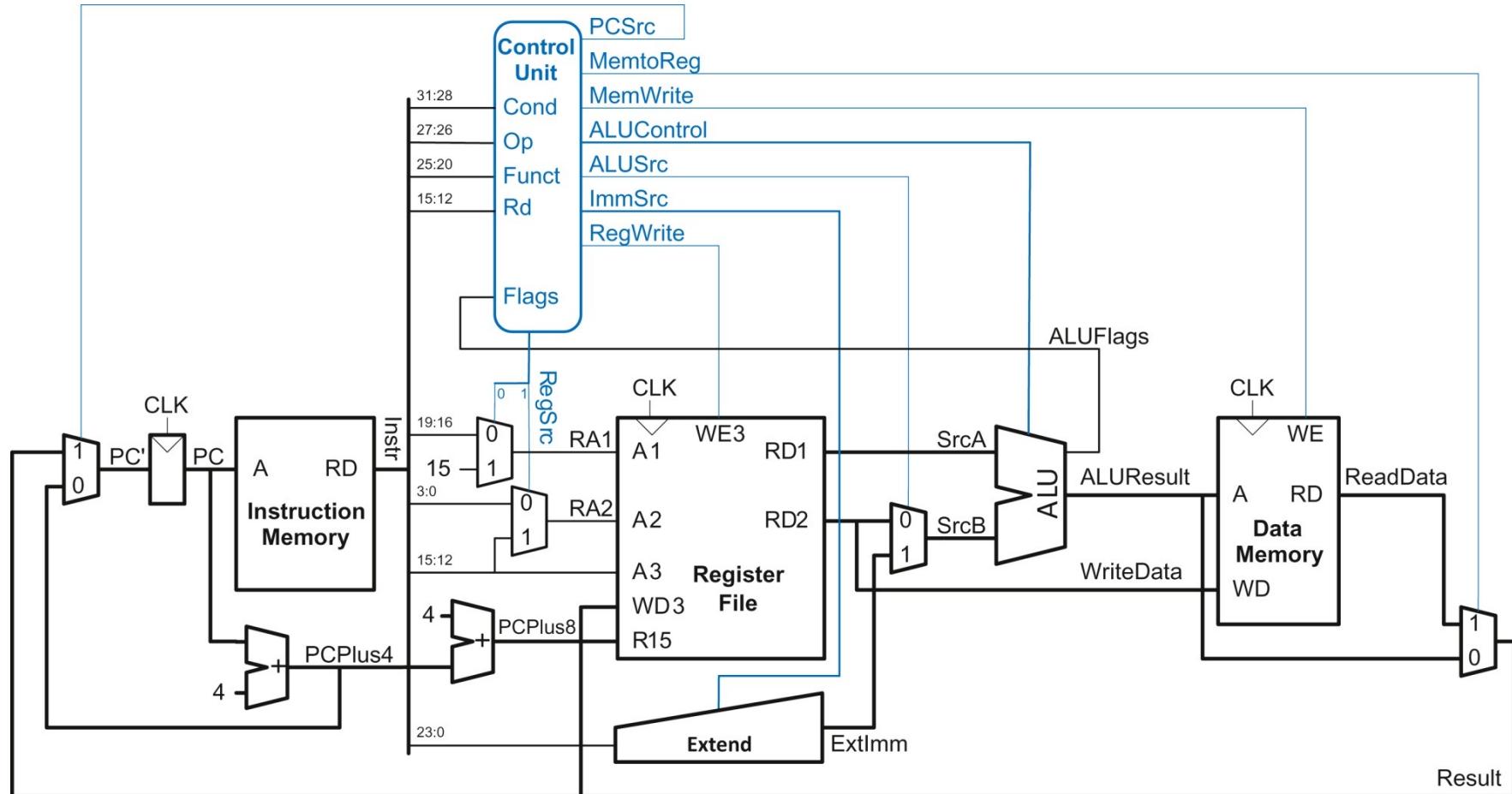
PCSrc	0
MemtoReg	0
MemWrite	0
ALUControl	11
ALUSrc	0
ImmSrc _{0:1}	XX
RegWrite	1
RegSrc _{0:1}	00

ALUControl	Function
00	ADD
01	SUB
10	AND
11	ORR

Processor Operation: ORR



Processor Operation: LDR



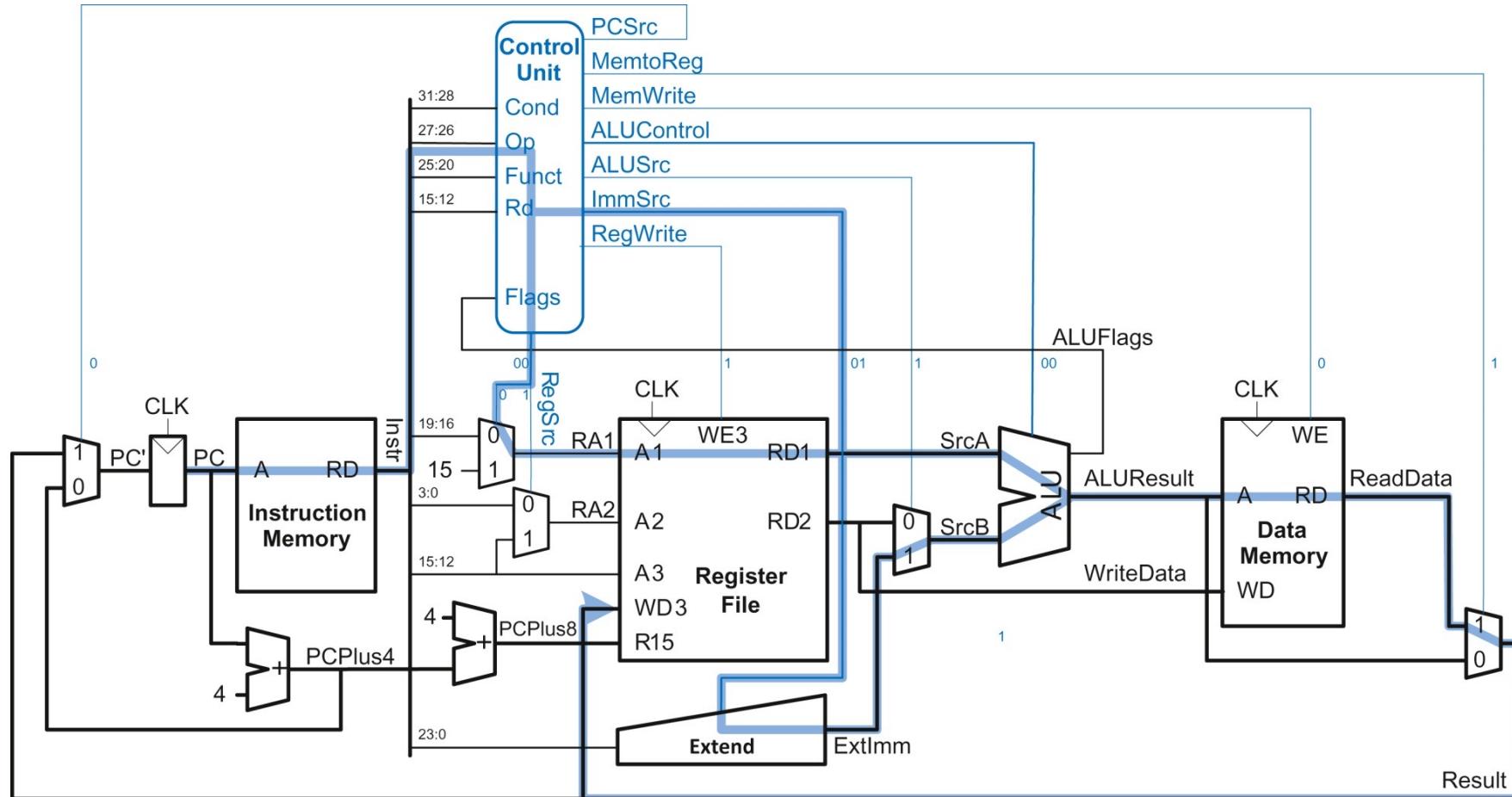
Processor Operation: LDR

PCSrc	0
MemtoReg	1
MemWrite	0
ALUControl	00
ALUSrc	1
ImmSrc _{0:1}	01
RegWrite	1
RegSrc _{0:1}	00

ALUControl	Function
00	ADD
01	SUB
10	AND
11	ORR

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} }	Sign-extended imm24

Processor Operation: LDR



Drawback of Single-Cycle CPU

- Is this the best way to build a CPU?
- What are the critical issues?
 - Next: performance analysis basics

Performance Analysis

Processor Performance

- Performance is **quantified** by the **execution time**
- The time it takes for a program to execute from start to finish
- For a given ISA and technology, how long does it take to run a program on the single-cycle CPU?

Processor Performance

- **How fast is my program?**
 - Every program consists of a series of instructions
 - Each instruction needs to be executed.
- **So how fast are my instructions ?**
 - Instructions are realized on the hardware
 - They can take one or more clock cycles to complete
 - *Cycles per Instruction = CPI*
- **How much time is one clock cycle?**
 - The critical path determines how much time one cycle requires = *clock period*.
 - $1/\text{clock period} = \text{clock frequency}$ = how many cycles can be done each second.

Execution Time

$$\text{Execution time} = (\text{\#instructions}) \left(\frac{\text{cycles}}{\text{instruction}} \right) \left(\frac{\text{seconds}}{\text{cycle}} \right)$$

- # instructions
 - Depends on the ISA, skill of programmer, compiler, algorithm
- cycles per instruction
 - Depends on the microarchitecture
- seconds per cycle (clock period, inverse is clock frequency, f)
 - critical path, circuit technology, type of adders, gate-level details

How can I Make the Program Run Faster?

- $N \times CPI \times (1/f)$
- **Reduce the number of instructions**
 - Make instructions that 'do' more (CISC)
 - Use better compilers
- **Use less cycles to perform the instruction**
 - Simpler instructions (RISC)
 - Use multiple units/ALUs/cores in parallel
- **Increase the clock frequency**
 - Find a 'newer' technology to manufacture
 - Redesign time critical components
 - Adopt pipelining

Execution Time (Single-Cycle CPU)

$$\text{Execution time} = (\text{\#instructions}) \left(\frac{\text{cycles}}{\text{instruction}} \right) \left(\frac{\text{seconds}}{\text{cycle}} \right)$$

- # instructions (**ARM is a RISC ISA**)
 - Depends on the ISA, skill of programmer, compiler, algorithm
- cycles per instruction (= **One, fixed, bad idea!**)
 - Depends on the microarchitecture
- seconds per cycle (**critical path of the CPU circuit**)
 - critical path, circuit technology, type of adders, gate-level details

Critical Path Analysis

- Each instruction in single-cycle CPU takes one clock cycle
- Determining the cycle time requires finding the critical path
- **Different instructions use different resources**
 - LDR uses instruction and data memory
 - ADD does not use data memory
 - STR does not write anything back to the register file
- **Which instruction is the slowest?**
 - Let us revisit the schematics and find out

Elements of Critical Path

Parameter	Description
t_{pcq_PC}	PC clock-to-Q delay
t_{mem}	Memory read
t_{dec}	Decoder propagation delay
t_{mux}	Multiplexer delay
t_{RFread}	Register file read
t_{ext}	Extension block delay
t_{ALU}	ALU delay
$t_{RFsetup}$	Set up RF for write (next cycle)

Critical Path: LDR

$$T_c = t_{pcq_PC} + t_{mem} + t_{dec} + \max[t_{mux} + t_{RFread}, t_{ext} + t_{mux}] + t_{ALU} \\ + t_{mem} + t_{mux} + t_{RFsetup}$$

- Memories & register files slower than combinational logic
 - Therefore, $t_{mux} + t_{RFread} \gg t_{ext} + t_{mux}$

Final Equation

$$T_c = t_{pcq_PC} + 2t_{mem} + t_{dec} + t_{RFread} + t_{ALU} + 2t_{mux} + t_{RFsetup}$$

Critical Path: DP-R

$$T_c = t_{pcq_PC} + t_{mem} + t_{dec} + t_{mux} + t_{RFread} + t_{mux} + t_{ALU} + t_{mux} \\ + t_{RFsetup}$$

Final Equation

$$T_c = t_{pcq_PC} + t_{mem} + t_{dec} + t_{RFread} + t_{ALU} + 3t_{mux} + t_{RFsetup}$$

Critical Path Analysis

- Different instructions have different critical paths
 - **LDR** is the **slowest** instruction
 - **DP–R** and **B** have **shorter critical paths** because they do not need to access data memory (**Memory is slow!**)
- Single-cycle processor is a **synchronous sequential circuit**
 - Clock period must be **constant** and **long enough** to accommodate the slowest instruction
- The numerical values of different variables in the critical path equation depend on the specific technology

Exercise: Perf Analysis

- Find the time it takes to execute a program with 100 billion instructions on a single-cycle CPU in 16 nm CMOS manufacturing process. See the table for delays of logic elements.

Parameter	Delay (ps)
t_{pcq_PC}	40
t_{mem}	200
t_{dec}	70
t_{mux}	25
t_{RFread}	100
t_{ALU}	120
$t_{RFsetup}$	60

$$T_c = t_{pcq_PC} + 2*t_{mem} + t_{dec} + t_{RFread} + t_{ALU} + 2*t_{mux} + t_{RFsetup}$$

Drawbacks of Single-Cycle CPU

- Requires two memories (no reuse)
- Requires three adders (no reuse)
- Clock period is dictated by the slowest instruction
 - No way to make the common case fast (DP instructions)

Multi-Cycle CPU

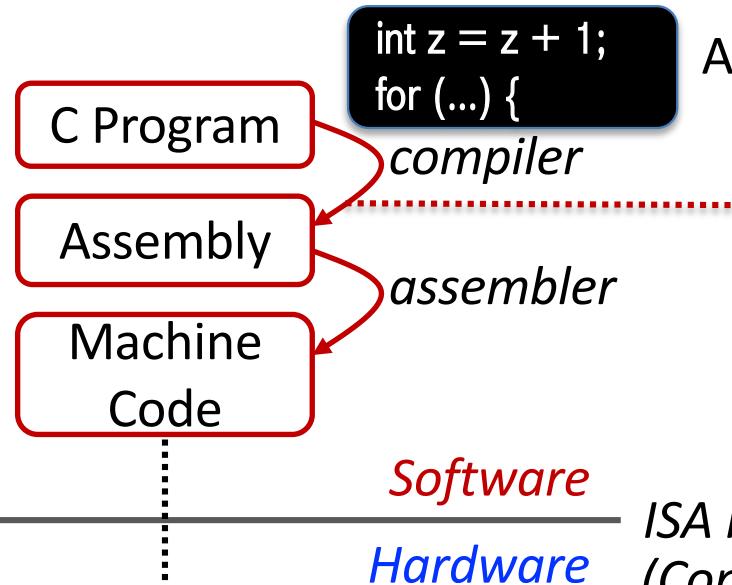
- Divide each instruction into a number of steps
- Perform one step in one clock cycle (instead of an entire instruction)
- Need **non-architectural (microarchitectural)** registers to store **intermediate state**
- Need an **FSM-based controller** to transition between steps
 - Different control signals on different steps

Section 7.4 of H&H

- **After the teaching break: Possible ext. for assignment 1**

COMP2300

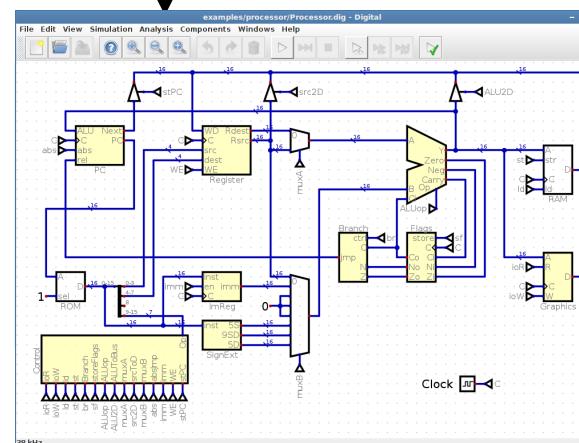
Application Software	>"hello world!"	Programs
Operating Systems		Device Drivers
Architecture		Instructions Registers
Micro-architecture		Datapaths Controllers
Logic		Adders Memories
Digital Circuits		AND Gates NOT Gates
Analog Circuits		Amplifiers Filters
Devices		Transistors Diodes
Physics		Electrons



Assignment 2: Program CPU



- ISA is the boundary
(Contract)



stored as 0's and 1's

*Fetch, decode, execute
an instruction every clock cycle*



Assignment 1: Build CPU

COMP2300

We will meet after two weeks

Revise the lecture content (DO ALL EXERCISES)

Finish assignment 1