CSCI 6461 Computer Systems Architecture

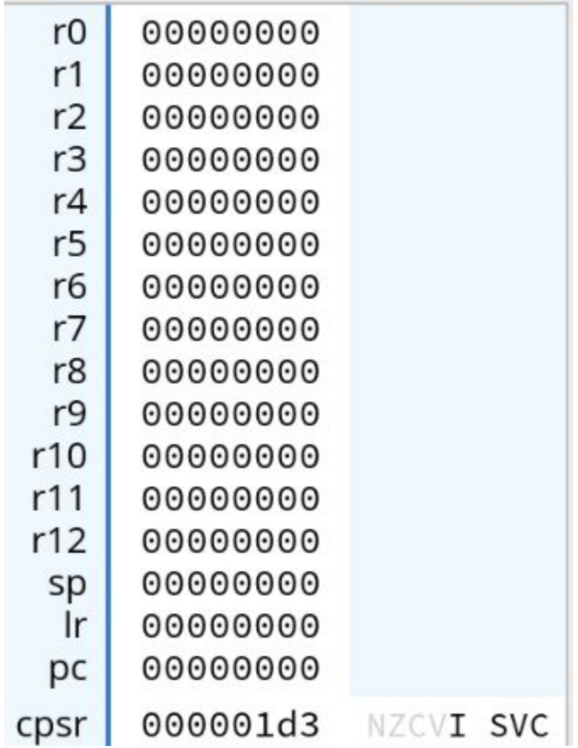
Single-Cycle Microarchitecture

Microarchitecture

- Microarchitecture is the specific hardware implementation of an instruction set architecture (ISA)
- It defines how instructions are executed through components like
  - datapaths,
  - control units, and
  - pipelines
- We will start with Single-Cycle Microarchitecture and move on to Pipeline Microarchitecture

Finite State Machine

- The architectural state for the ARM32 consists of 16 32-bit registers and the status register.
    - `r0-r15`, where `r13` is `sp`, `r14` is `lr`, and `r15` is `pc`
- A microarchitecture must contain all of its **state**.
- Based on the current architectural state, the processor executes a particular instruction with a set of data to produce a new architectural state
- This is a type of Finite State Machine (**FSM**)



```
  r0   00000000
  r1   00000000
  r2   00000000
  r3   00000000
  r4   00000000
  r5   00000000
  r6   00000000
  r7   00000000
  r8   00000000
  r9   00000000
 r10   00000000
 r11   00000000
 r12   00000000
  sp   00000000
  lr   00000000
  pc   00000000

cpsr   000001d3   NZCVI SVC
```

Single-Cycle microarchitecture

- Single-cycle microarchitecture executes instructions in a **single cycle** .
  - fetch/decode/execute/write-back complete **within** the cycle
  - no pipeline => the clock period must be long enough to accommodate the slowest instruction
- Each instruction takes the same time (in reality they don't)
- State updates made at the end of an instruction’s execution
- The slowest instruction determines cycle time
  - => long clock cycle times
  - this is a disadvantage

Instruction subset

To keep the microarchitectures easy to understand, we consider only a subset of the ARM instruction set:

- Memory instructions: `ldr`, `str` (with positive immediate offset)
- Data-processing instructions: `add, sub, and, orr`
- Branches: `b`
- We begin with `ldr`, `str` because, as the components used memory instructions can be reused in data processing instructions

The microarchitecture **datapath**

Datapath

- We divide our microarchitectures into two interacting parts:
  - the datapath and
  - the control unit
- The datapath operates on words of data
- It contains structures such as
  - memories
  - registers
  - ALUs, and
  - multiplexers.

Control Unit

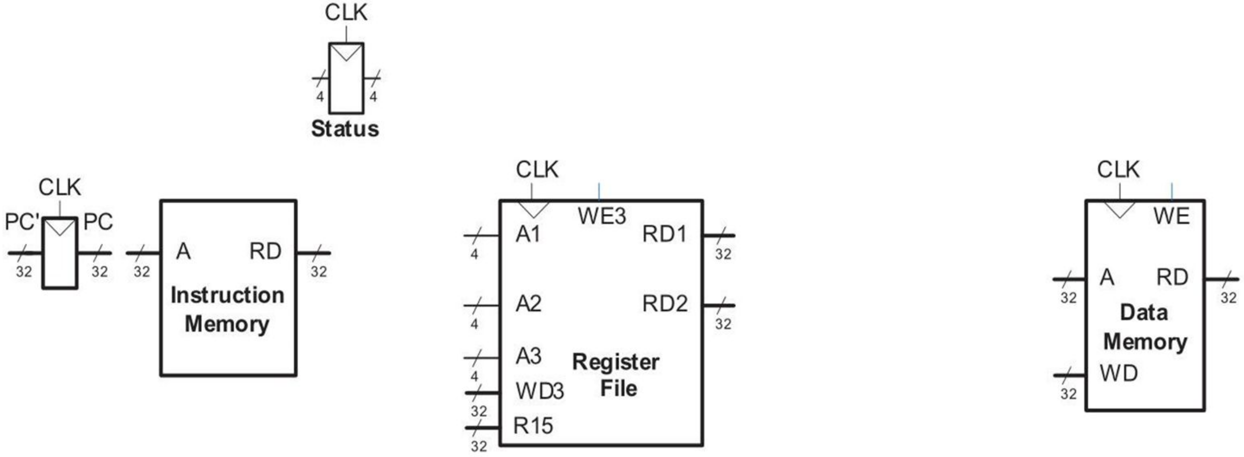
- The control unit receives the current instruction from the datapath
  - and tells the datapath how to execute that instruction.
- The control unit produces
  - multiplexer select
  - register enable and
  - memory write signals
- This controls the operation of the datapath.

General Approach

- We begin with hardware containing the **state elements** .
- These elements include the memories and the architectural state
  - the program counter
  - registers, and
  - status register
- Then, we add blocks of combinational logic between the state elements to compute the new state based on the current state.

Single-cycle microarchitecture

- We begin constructing the datapath by connecting the state elements with combinational logic that can execute the various instructions.



A note on the pc

- Although the program counter (pc) is logically part of the register file
- it is read and written on every cycle independent of the normal register file operation
- it's more naturally built as a stand-alone 32-bit register.
- Its output, pc, points to the current instruction.
  - Its input, pc` indicates the address of the next instruction

Adding state elements

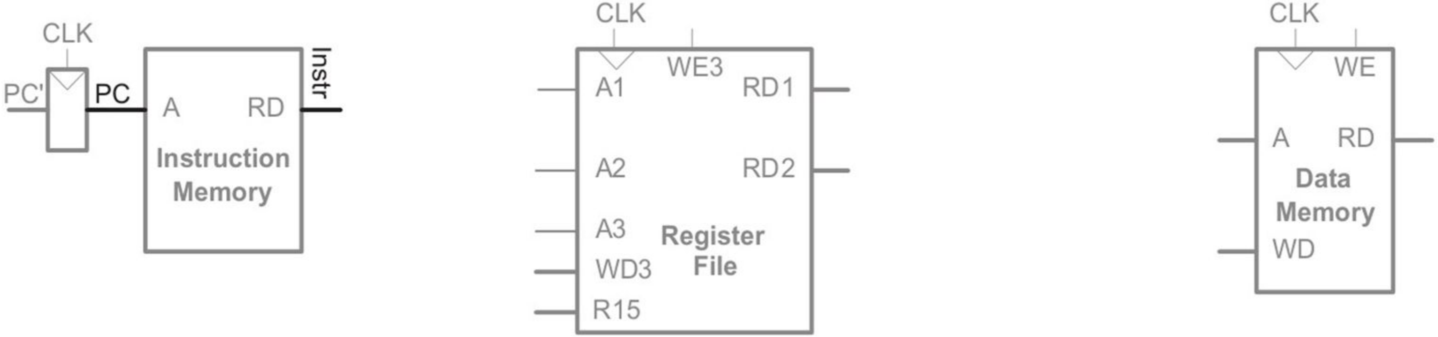
- Control signals determine which specific instruction is performed by the datapath at any given time.
- The control unit contains combinational logic that generates the appropriate control signals
    - based on the current instruction.
- We will gradually develop the single-cycle datapath, adding one piece at a time to the state elements.

Adding state elements

- The new connections are emphasized in black (or blue, for new control signals),
- whereas the hardware that has already been studied is shown in gray.
- The status register is part of the controller
    - and will be omitted while we focus on the datapath.

First Step

- The program counter contains the address of the instruction to execute.
- The first step is to read this instruction from instruction memory.



PC connected

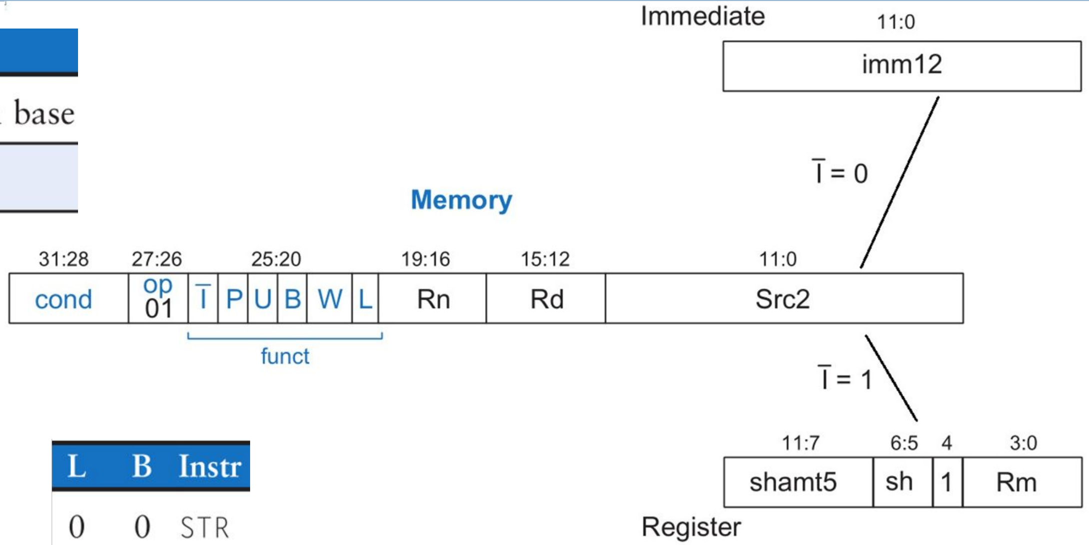
- The `pc` is connected to the address input of the instruction memory.
- The instruction memory reads out, or fetches, the 32-bit instruction, labeled **Instr** .
- The processor’s actions depend on the specific instruction that was fetched.
- Lets look at `ldr/str` operations first

`ldr` Implementation

Example: `ldr r7, [r5, #8]`

Recap: Memory Instructions

Bit	$\bar{I}$	U
0	Immediate offset	Subtract offset from base
1	Register offset	Add offset to base



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

Immediate ($\bar{I}$ = 0)

11:0
imm12

Register ($\bar{I}$ = 1)

11:7	6:5	4	3:0
shamt5	sh	1	Rm

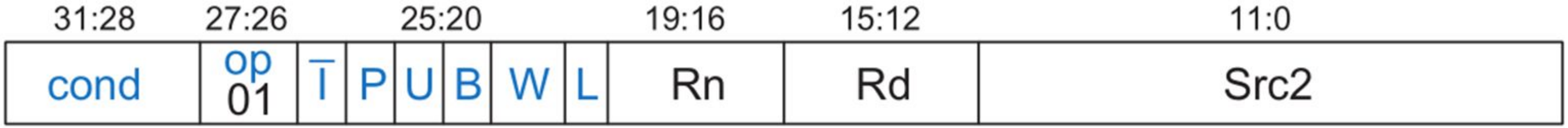
P	W	Index Mode
0	0	Post-index
0	1	Not supported
1	0	Offset
1	1	Pre-index

L	B	Instr
0	0	STR
0	1	STRB
1	0	LDR
1	1	LDRB

The offset is either a 12-bit unsigned immediate imm12 or a register Rm that is optionally shifted by a constant shamt5.

`ldr` Example

- We start with `ldr` with **positive immediate offset** .
- For example, `ldr r7, [r5, #8]`
- For the `ldr` instruction, the next step is to read the source register containing the base address.
- This register is specified in the **Rn** field of the instruction.



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

The bits 19:16 of the instruction are connected to the address input of one of the register file ports, A1.

`ldr r7, [r5, #8]` Hex and Binary

Lets look at the hex and binary for

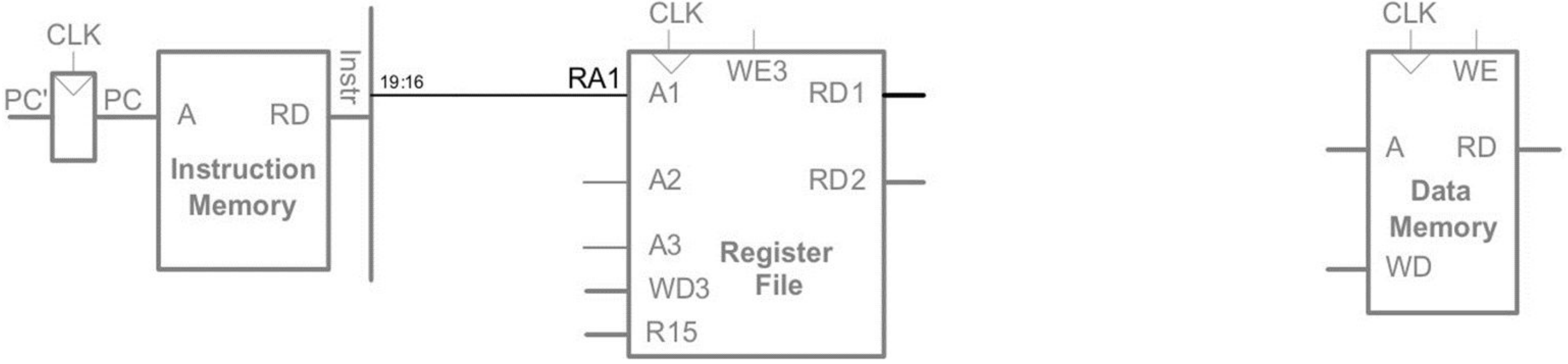
```
ldr r7, [r5, #8]
```

using CPUlator to generate the hex and then https://www.rapidtables.com/convert/number/hex-to-binary.html to convert to binary:

Example `ldr r7, [r5, #8]`

- `ldr r7, [ r5, #8 ]` = 0xe5957008
- 1110 = undonditional
- 01 = Memory Operation
  - 0 1 1 0 0 1
  - I P U B W L
  - Immediate (I bar), Offset (P/W), LDR (L/B), Add to base (U)
- 0101 = r5
- 0111 = r7
- 000000001000 = #8

19:16(Rn) `ldr r7, [r5, #8]`



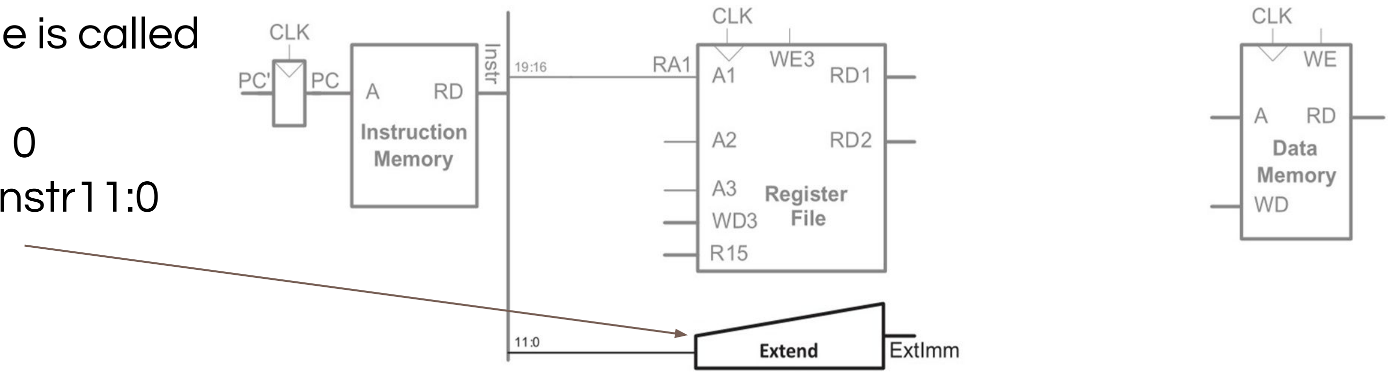
So the register address of the `ldr (r5)` is passed into the register file. The register file reads the register value onto RD1.

Zero extending `ldr r7, [r5, #8]`

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

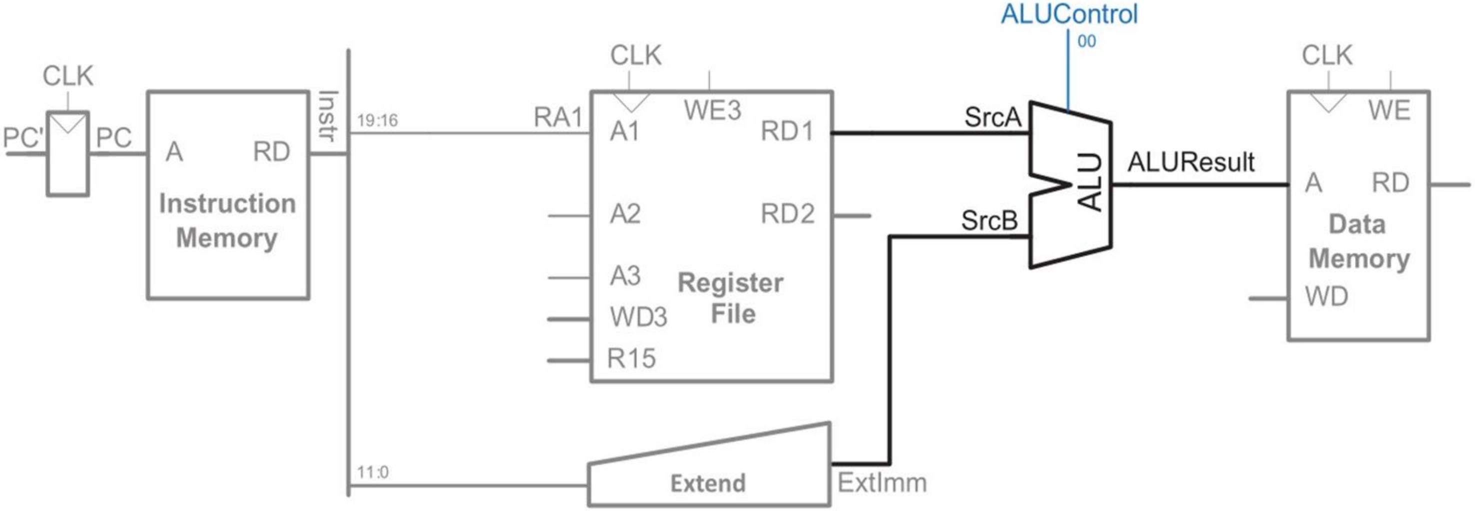
The offset, stored in the bits 11:0, is an unsigned value, so it must be zero-extended to 32 bits.

The 32-bit value is called ExtImm
ImmExt31:12 = 0
ImmExt11:0 = Instr11:0



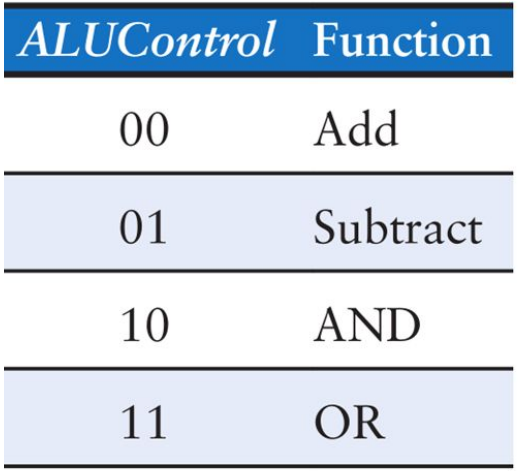
ALU Addition `ldr r7, [ r5, #8 ]`

The processor must add the base address to the offset to find the address to read from memory.



ALU Operations `ldr r7, [r5, #8]`

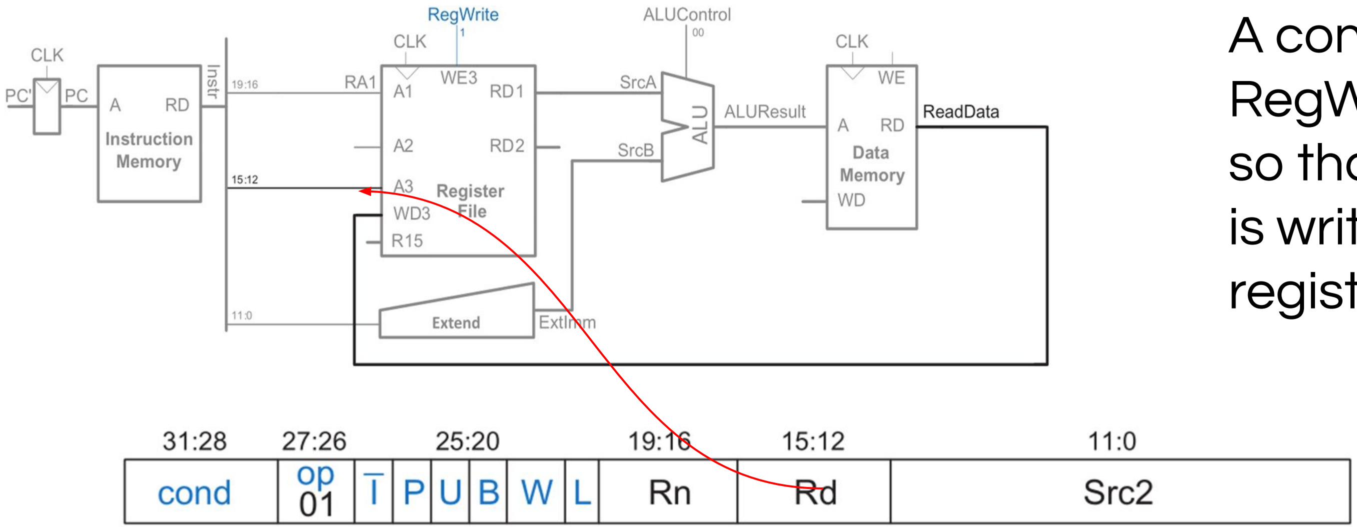
- For an LDR instruction, ALUControl should be set to 00 to perform addition
- ALUResult is sent to the data memory address to read.



ALUControl	Function
00	Add
01	Subtract
10	AND
11	OR

Read data memory `ldr r7, [r5, #8]`

The data is read from the data memory onto the ReadData bus and then written back to the destination register on the rising edge of the clock at the end of the cycle.

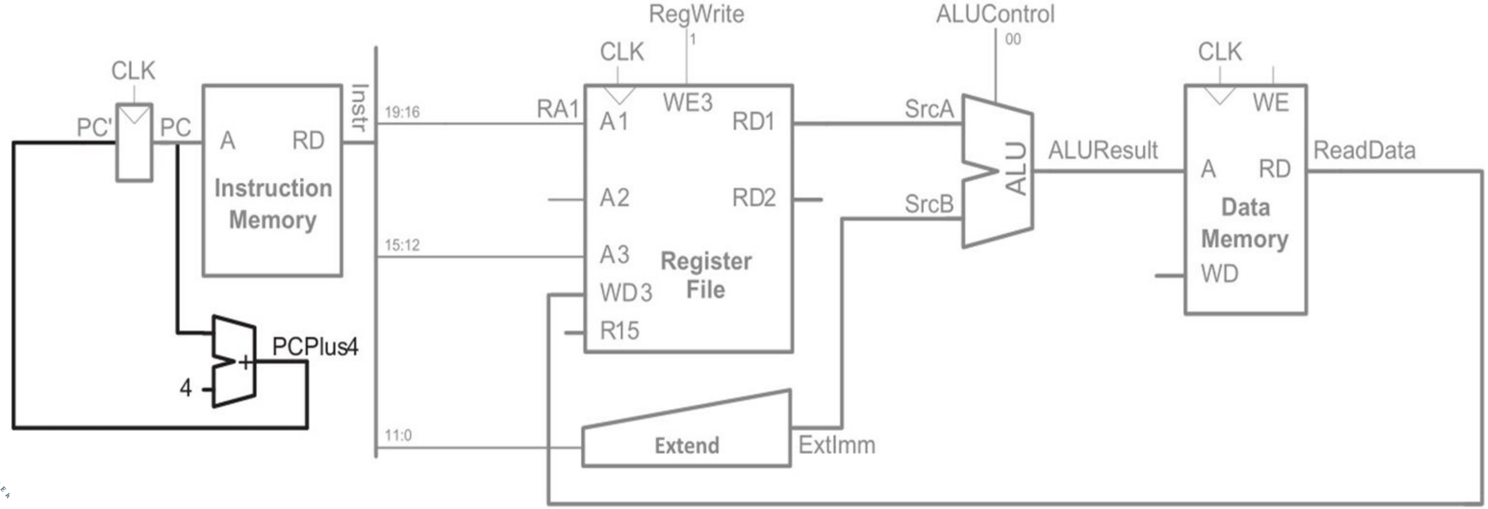


31:28	27:26	25:20						19:16	15:12	11:0
cond	op 01	Ī	P	U	B	W	L	Rn	Rd	Src2

A control signal RegWrite is asserted so that the data value is written into the register file.

Next, increment PC'

While the instruction is being executed, the processor must compute the address of the next instruction, PC'



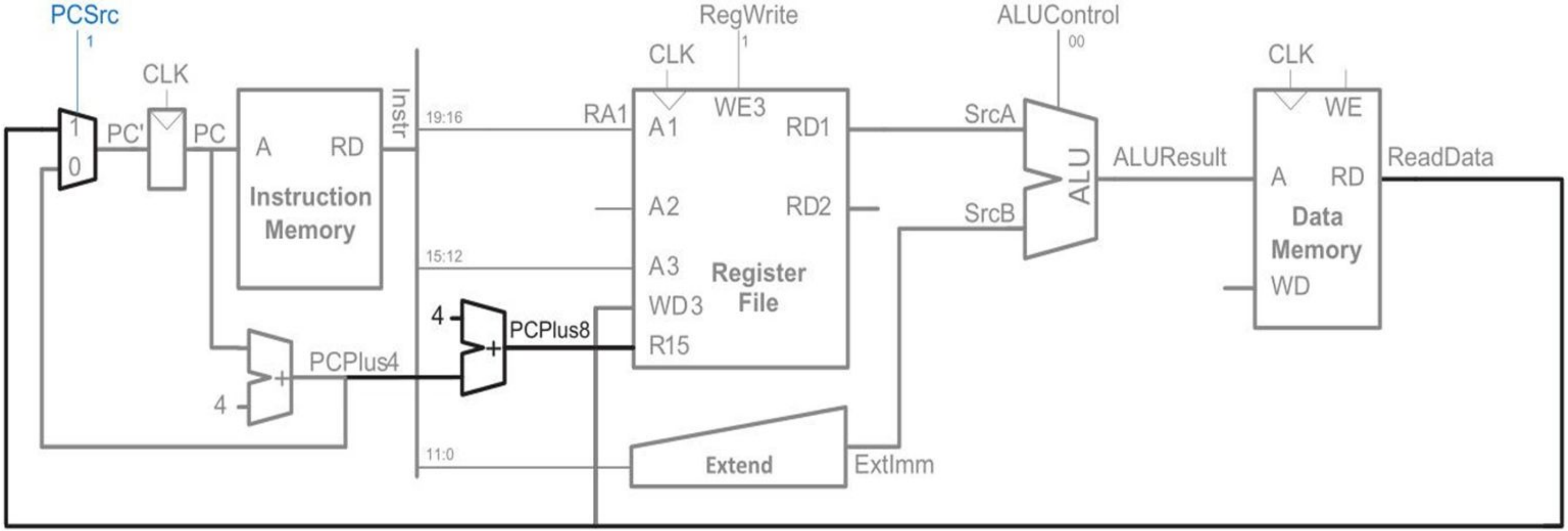
`ldr` … finished

- The new address is written into the program counter on the next rising edge of the clock.
- This completes the datapath for the LDR instruction, except for the case of the base or destination register being R15.
- Note the increment applied to PC, +4 is added via the PCPlus4 block
    - While the instruction is being executed
- We will look at that case next

`r15` Read

- In the ARM architecture, reading register R15 returns PC+8
- Another adder further increments the PC by 4.
- Writing register R15 updates the PC.
- PC' may come from the result of the instruction (ReadData) rather than PCPlus4.
- The PCSrc control signal is set to 0 to choose PCPlus4 or 1 to choose ReadData.
    - for example, `pop {pc}` requires a **ReadData**

Update the PC



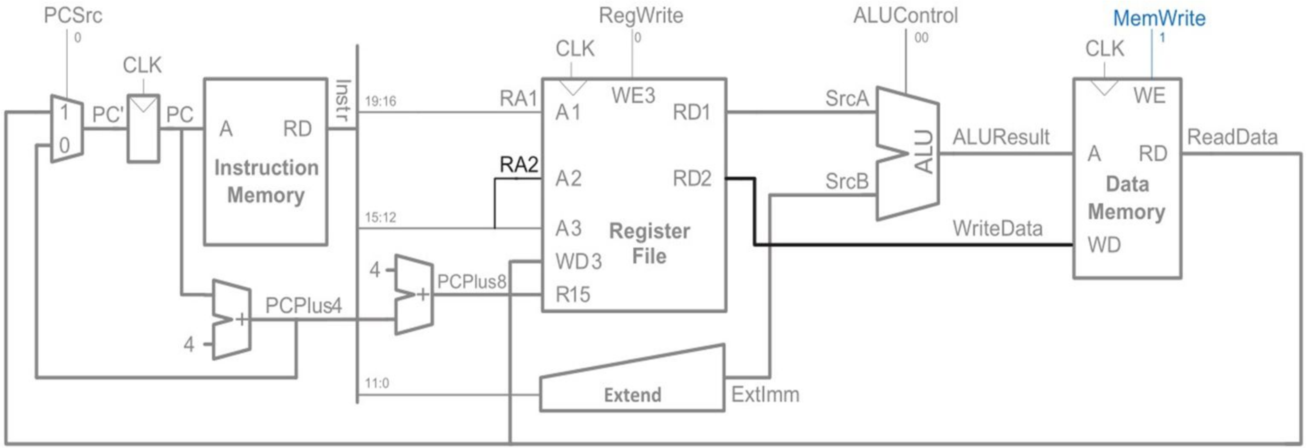
Extension to STR

Extend the previous example

`str` example

- Lets consider `str r7, [r5, #8]`
- Like `ldr`, `str` reads a base address from port 1 of the register file and zero-extends the immediate.
- The ALU adds the base address to the immediate to find the memory address.
- All of these functions are already supported in the datapath.
- The `str` instruction also reads a second register from the register file and writes it to the data memory.
- The Rd value is read onto the RD2 port **via RA2** .

Extension to `str r7, [r5, #8]`



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

MemWrite=1 to write the data to memory; ALUControl=00 to add the base address and offset; RegWrite=0, because nothing should be written to the register file.

Finally

- Note that data is still read from the address given to the data memory
- Note again - RA2 is chosen from the Rd field (Instr15:12) for STR and the Rm field (Instr3:0) for data-processing instructions with register addressing based on the RegSrc control signal
- But that this ReadData is ignored
  - because RegWrite = 0.

Datapath enhancements

for `add`, `sub`, `and`, `orr`

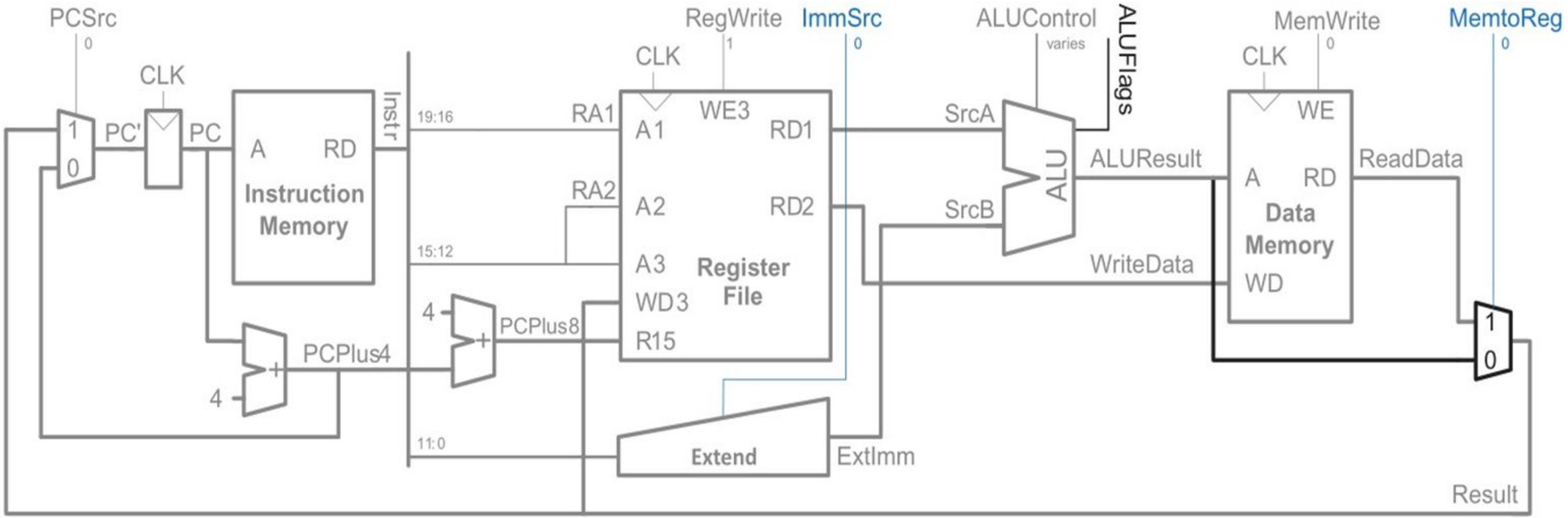
`add`, `sub`, `and`, `orr`

- We now extend the datapath to handle the data-processing instructions,
  - `add`, `sub`, `and`, and `orr`,
  - using the immediate addressing mode.
  - Example: `add r1, r2, #3`
- `add r1, r2, #3` reads a source register from the register file and an immediate from the low bits of the instruction
- performs ALU Add operation on them,
  - and writes the result back to register Rd (R1)

`add`, `sub`, `and`, `orr`

- These instructions are similar
- Hence, they can all be handled with the same hardware
- But using different **ALUControl** signals.
- The ALU also produces four flags, ALUFlags 3:0 (Negative, Zero, Carry, oVerflow),
  - that are sent back to the controller.

Example `add r1, r2, #3`



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

ImmSrc role

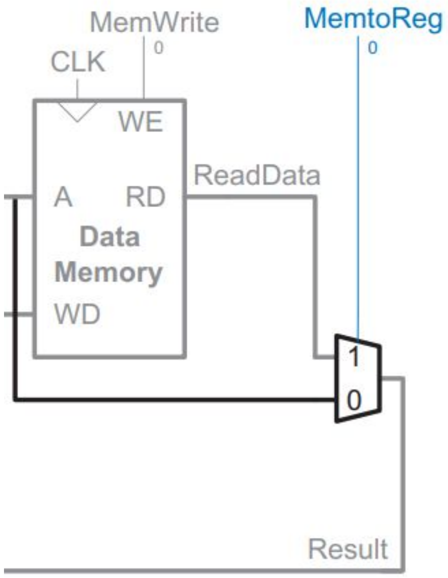
- Data-processing instructions use only an 8-bit immediate rather than a 12-bit immediate.
- Therefore, we provide the ImmSrc control signal to the Extend block.
- When it is 0,
  - ExtImm is zero-extended from Instr7:0 for data-processing instructions.
- When it is 1,
  - ExtImm is zero-extended from Instr11:0 for `ldr` or `str`.

Additional Multiplexer

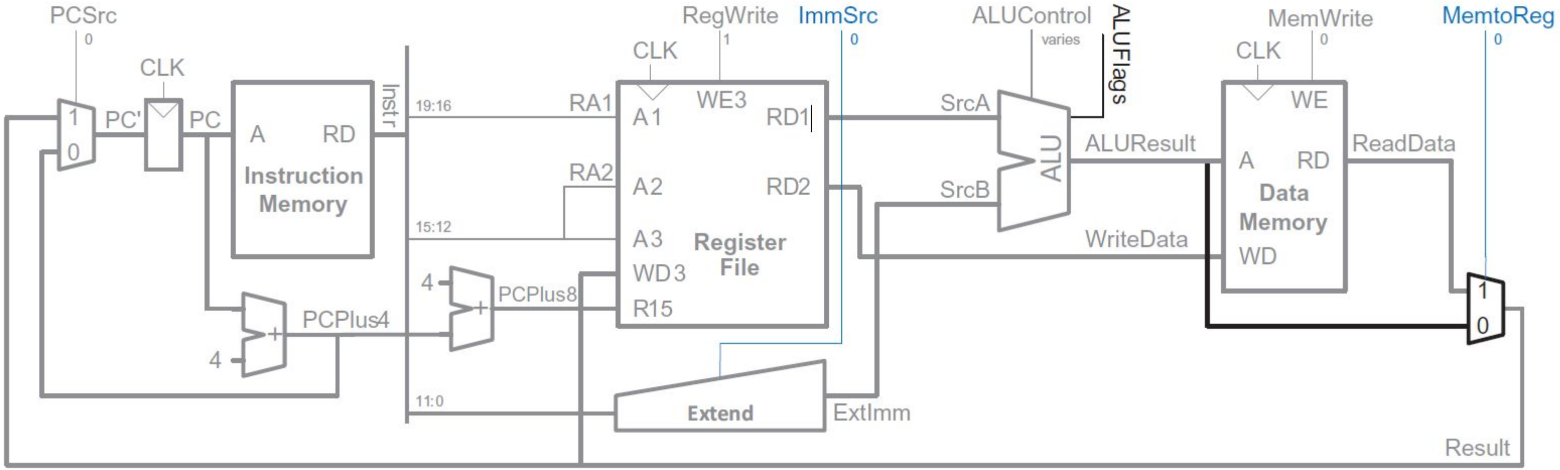
- For `ldr`, the register file received its write data from the data memory.
- However, data-processing instructions write ALUResult to the register file.
- Therefore, we add another multiplexer to choose between ReadData and ALUResult.
- We call its output **Result** .

MemtoReg Multiplexer

- The multiplexer is controlled by another new signal, MemtoReg.
- MemtoReg is 0 for data processing instructions to choose Result from ALUResult;
- it is 1 for `ldr` to choose ReadData.



Finally



The microarchitecture for `add r1, r2, #3`

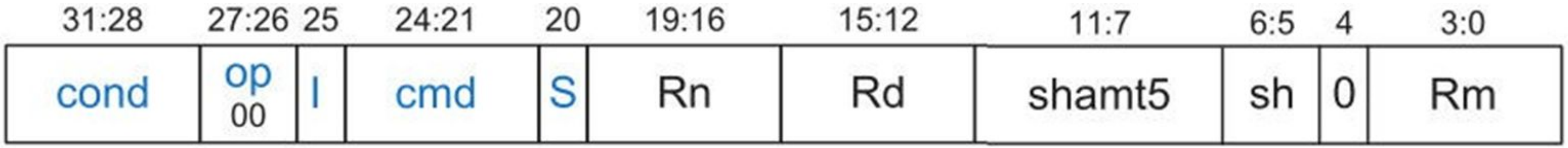
31:28	27:26	25:20	19:16	15:12	11:0
cond	op	funct	Rn	Rd	Src2
4 bits	2 bits	6 bits	4 bits	4 bits	12 bits

DP and Register Addressing

```
add r1, r2, r3
```

Example: `add r1, r2, r3`

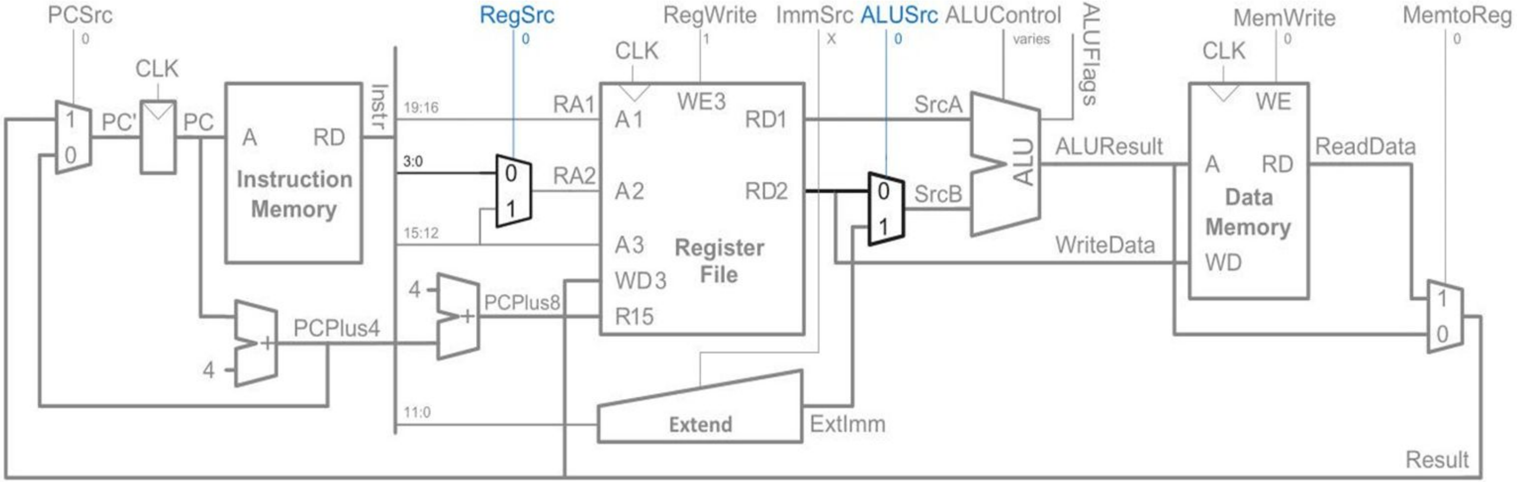
- Example: `add r1, r2, r3`
- Data-processing instructions with register addressing receive their 2nd source from **Rm**, specified by Instr3:0



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

We must add multiplexers on the inputs of the register file and ALU to select this second source register.

Multiplexers - **RegSrc** and **ALUSrc**



RegSrc - Select from Instr15:12 for STR and the **Rm** field (Instr3:0) for data-processing. **ALUSrc** select between ExtImm or RD2

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

RegSrc and ALUSrc

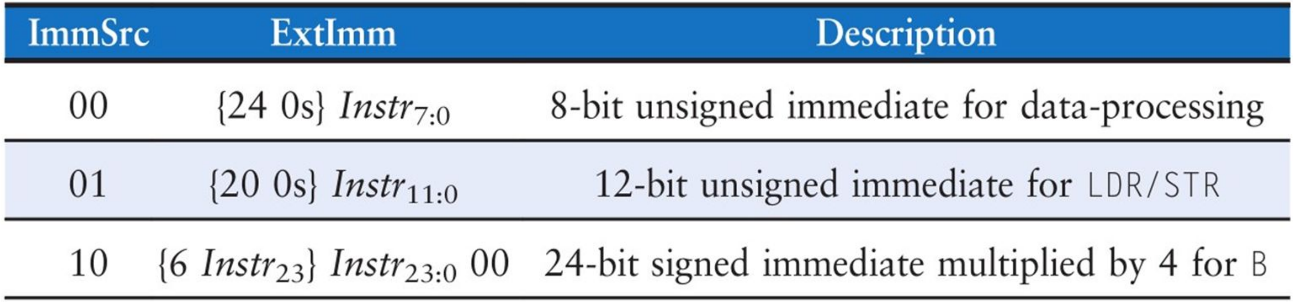
- RA2 is chosen from
  - Rd field (Instr15:12) for STR and
  - Rm field (Instr3:0) for data-processing instructions with register addressing
- based on the **RegSrc** control signal
- Based on the **ALUSrc** control signal, the second source to the ALU is selected from ExtImm for instructions using immediates
  - and from the register file for data-processing instructions with register addressing.

Data processing & Branching

The `b` instruction

Immsrc and the b instruction

- The branch instruction adds a 24-bit immediate to PC+8
  - and writes the result back to the PC.
- The immediate is multiplied by 4 and sign extended.
- Therefore, the Extend logic needs yet another mode. **ImmSrc** is increased to 2 bits



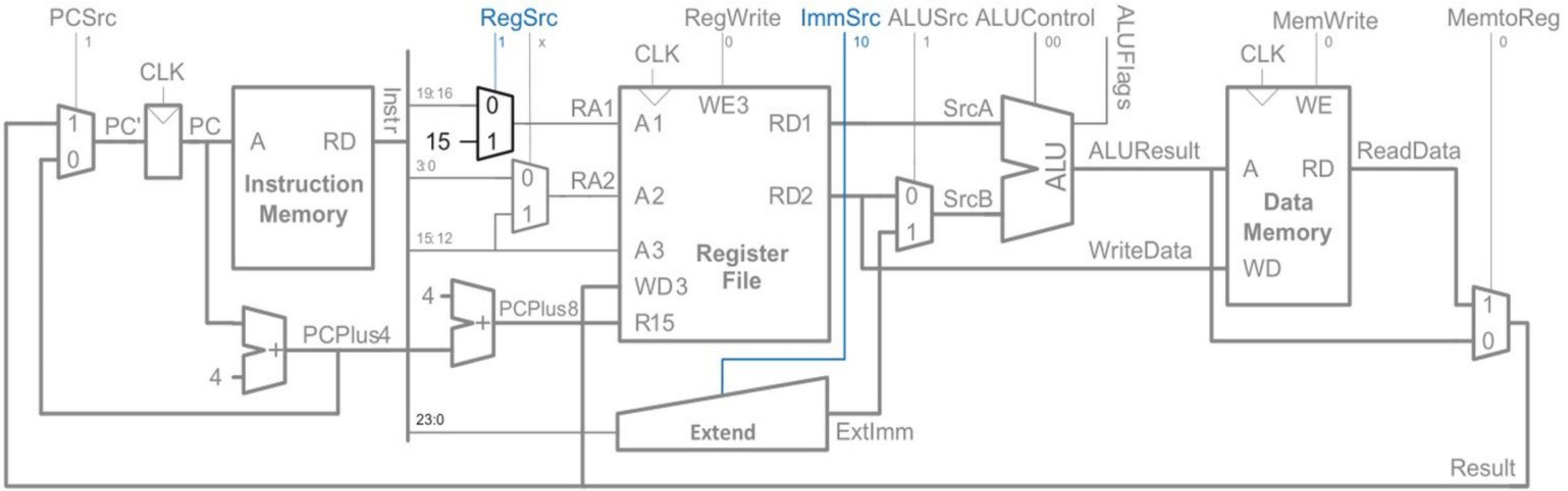
ImmSrc	ExtImm	Description
00	{24 0s} $Instr_{7:0}$	8-bit unsigned immediate for data-processing
01	{20 0s} $Instr_{11:0}$	12-bit unsigned immediate for LDR/STR
10	{6 $Instr_{23}$} $Instr_{23:0}$ 00	24-bit signed immediate multiplied by 4 for B

Signed immediate extension

The code $\{6\ \texttt{Instr}_{23}\}\ \texttt{Instr}_{23:0}\ \texttt{00}$ is understood in two steps

- The shit left by 2 means the 2 LSB are zero filled
- And the remaining space (6 bits) get their value from the $\texttt{Instr}_{23}$ field - which of course is sign extended
- This effectively gives an addressable range of
  - Range = ± (2^{25}) bytes = ± 32 MB
  - Because we need one bit for the sign (hence the 2^{25})

Datapath with Branching



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

Note that RegSrc now becomes a two-bit control signal and is connected to two multiplexers as discussed next

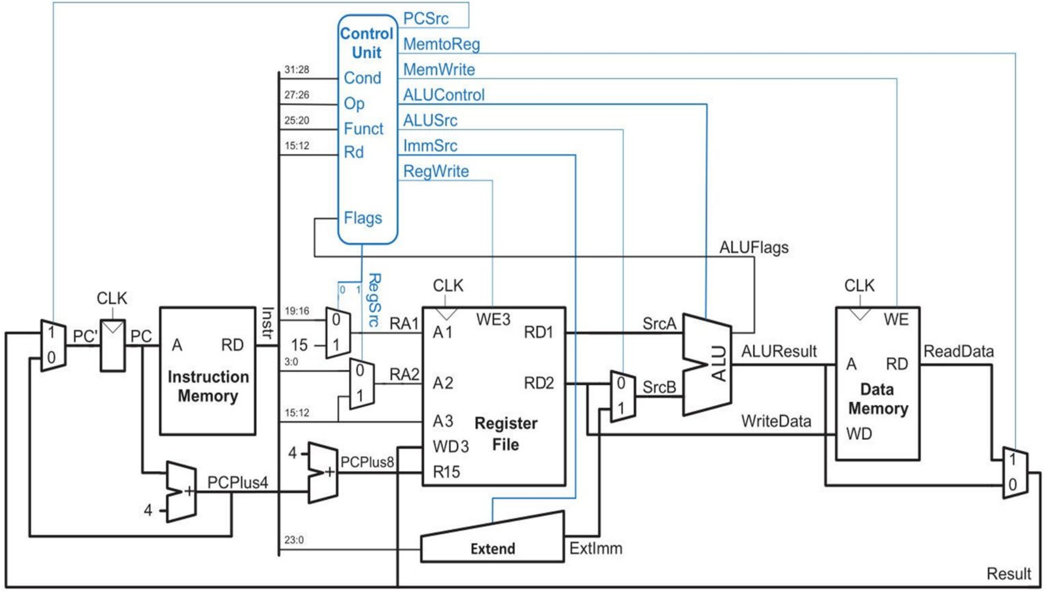
Datapath with Branching

- PC+8 is read from the first port of the register file.
- Therefore, a multiplexer is needed to choose R15 as the RA1 input.
- It is controlled by another bit of **RegSrc**, choosing `Instr`$_{19:16}$ for most instructions but 15 for `b`
- MemtoReg is set to 0 and PCSrc is set to 1 to select the new PC from ALUResult for the branch
- This is `pc`-relative addressing as mentioned

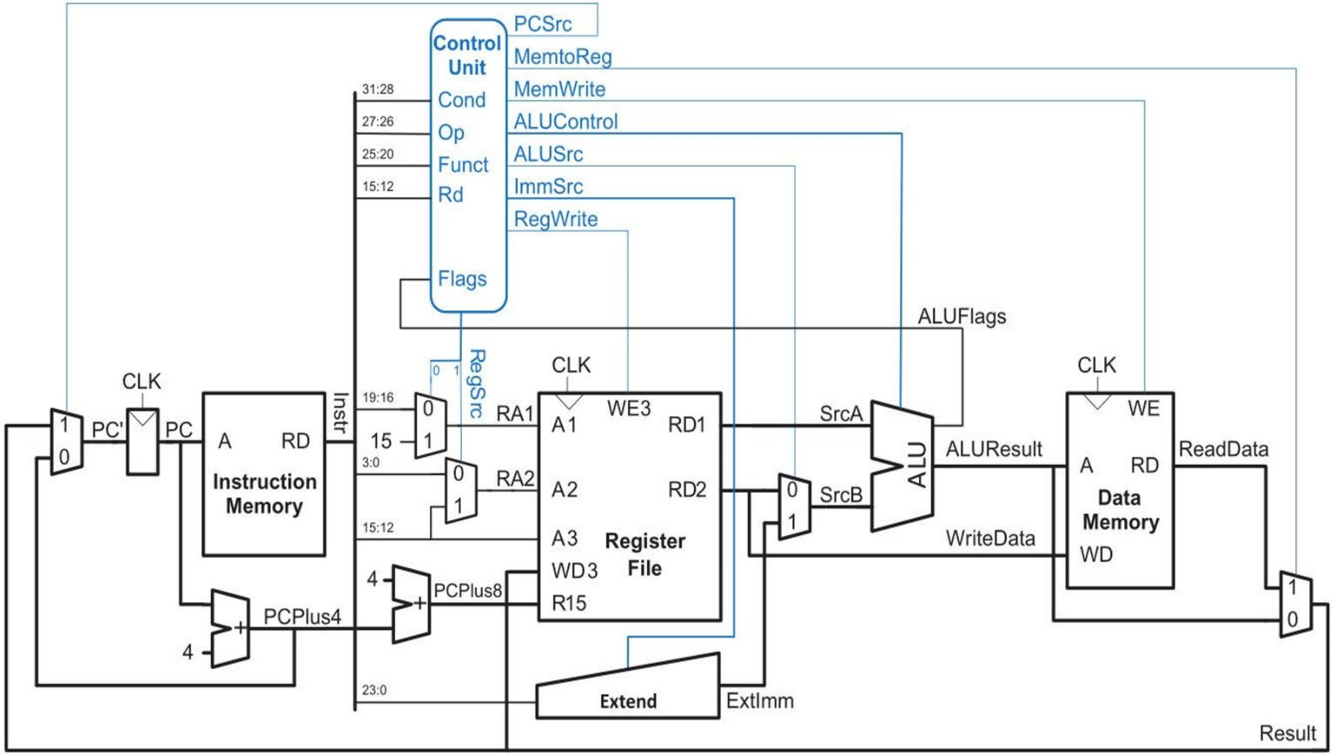
Control Unit

Control Unit

The control unit computes the control signals based on the **cond, op, funct** fields of the instruction ($\text{Instr}_{31:28}$, $\text{Instr}_{27:26}$, $\text{Instr}_{25:20}$) as well as the flags and whether the destination register is the PC.



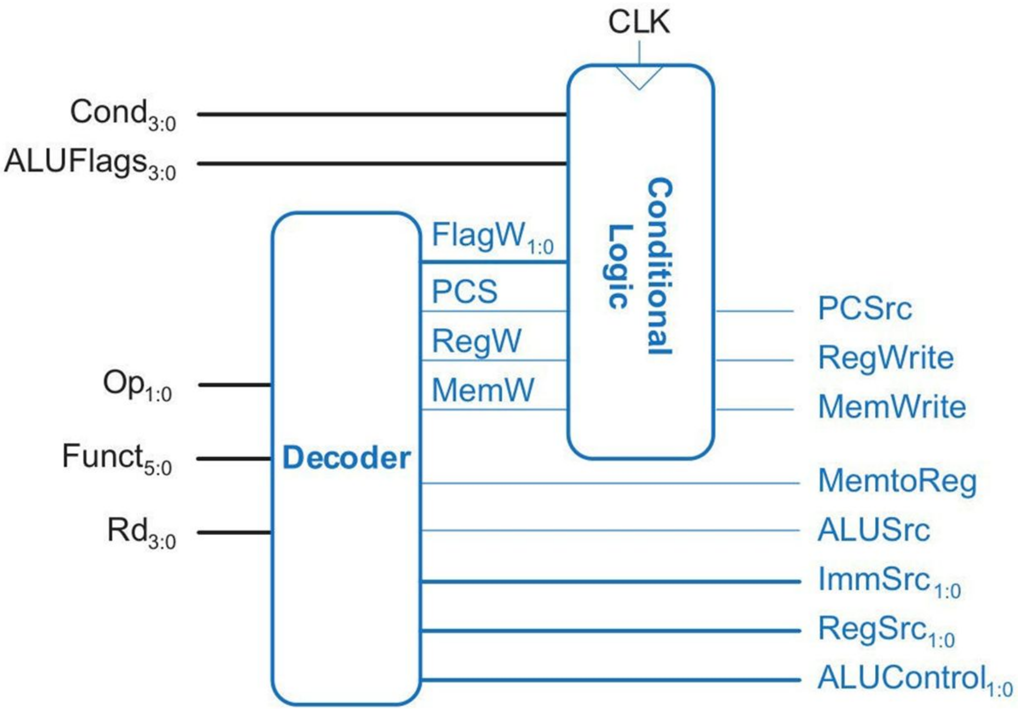
Control Unit



Conditional Logic

The conditional logic maintains the status flags and only enables updates to architectural state when the instruction should be conditionally executed.

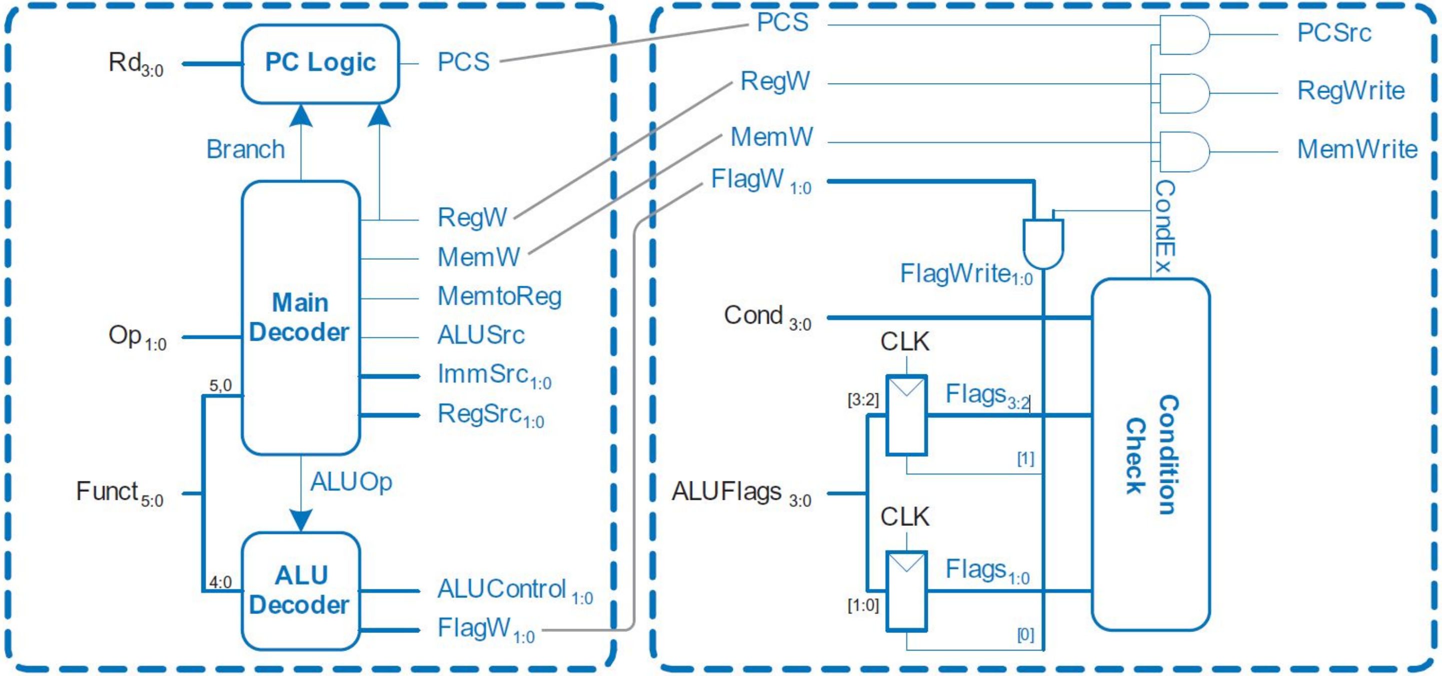
The decoder generates control signals based on Instr.



Main decoders

- The main decoder produces most of the control signals.
- The ALU decoder uses the **funct** field to determine the type of data-processing instruction.
- The PC Logic determines whether the PC needs updating due to a branch or a write to R15.

Decoder and Conditional Logic



Decoder and Conditional Logic

The controller is in two main parts:

- the **Decoder** which generates control signals based on Instr, and the
- **Conditional Logic** which maintains the status flags and only enables updates to architectural state when the instruction should be conditionally executed
- The **Conditional Logic** , determines whether the instruction should be executed (CondEx) based on the cond field and the current values of the N, Z, C, and V

ALUSrc

- ALUSrc control signal:
  - the second source to the ALU is selected from ExtImm for instructions using immediates (ALUSrc=1)
  - and from the register file for data-processing instructions with register addressing (ALUSrc=0)

Register Source: **RegSrc**

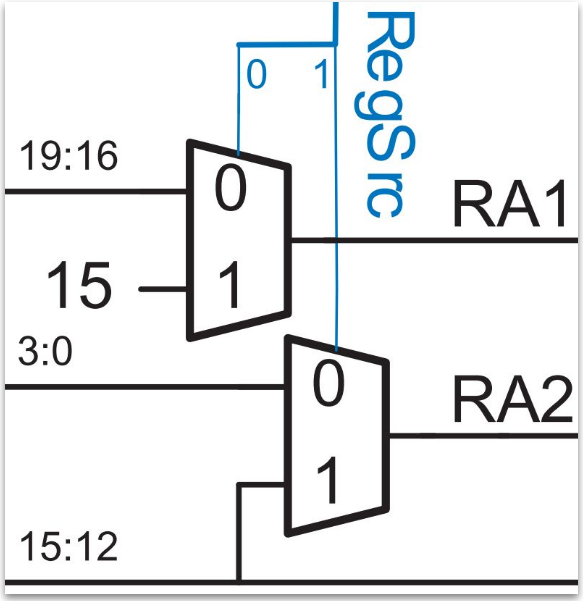
In the following examples we consider only these operations:

- DP with Regs
- DP with Immediates
- LDR with Immediates
- STR with Immediates
- Branching

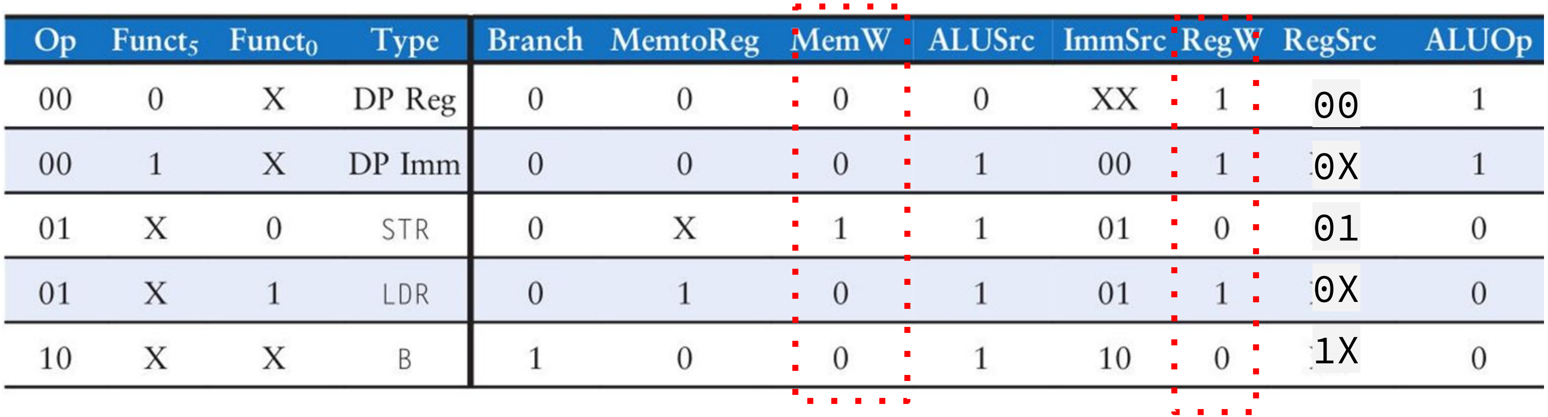
For the sake of simplicity, LDR/STR with registers is omitted, which in turn simplifies the **RegSrc** control path

RegSrc bit ordering (simplified)

- `00`: RA1 gets 19:16, RA2 gets 3:0
  - DP with Regs
- `0X`: RA1 gets 19:16, RA2 don't care
  - DP with Imm
  - LDR with Imm
- `01`: RA1 gets 19:16, RA2 gets 15:12
  - STR with Imm (2 reads)
- `1X`: RA1 gets R15, RA2 don't care
  - Branch



Main decoder truth table

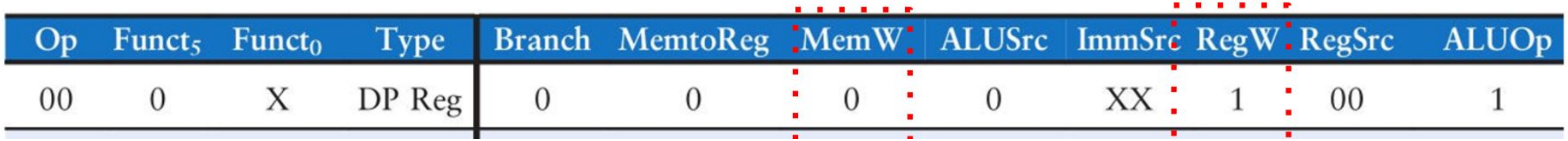


Op	$Funct_5$	$Funct_0$	Type	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp
00	0	X	DP Reg	0	0	0	0	XX	1	00	1
00	1	X	DP Imm	0	0	0	1	00	1	0X	1
01	X	0	STR	0	X	1	1	01	0	01	0
01	X	1	LDR	0	1	0	1	01	1	0X	0
10	X	X	B	1	0	0	1	10	0	1X	0

Where op: 00 = DP, 01= Memory, 10 = B

The main decoder determines the type of instruction: data-processing register, data-processing immediate, STR , LDR , B. It sends **MemtoReg** , **ALUSrc** , **ImmSrc1:0** , and **RegSrc1:0** directly to the datapath.

DP Example (op=00): `add r0,r1,r2`



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

funct $_5$ (I) = 0: Register Value, not immediate

funct $_0$ (S) = X: don't care

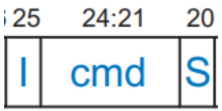
Branch = 0: no branch

MemtoReg = 0: this is RegW

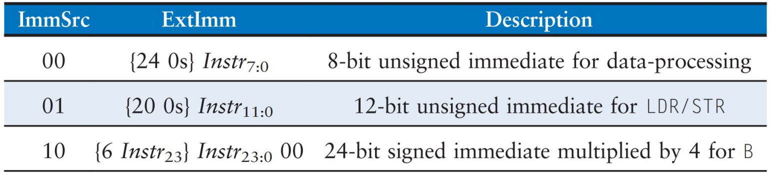
ALUSrc = 0: Use RD2

ImmSrc = XX: don't care

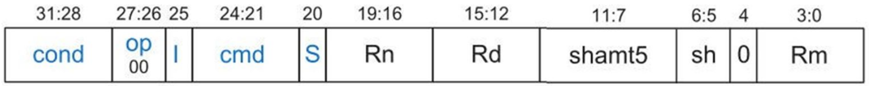
RegSrc = 00: 19:16 and 3:0



25	24:21	20
I	cmd	S

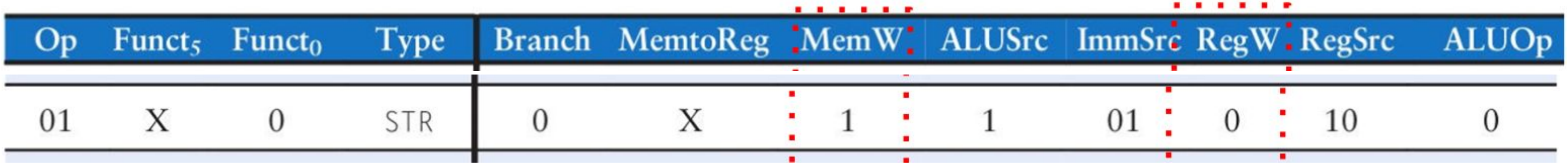


ImmSrc	ExtImm	Description
00	{24 0s} $Instr_{7:0}$	8-bit unsigned immediate for data-processing
01	{20 0s} $Instr_{11:0}$	12-bit unsigned immediate for `LDR`/`STR`
10	{6 $Instr_{23}$} $Instr_{23:0}$ 00	24-bit signed immediate multiplied by 4 for `B`



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

(op=01) str with positive immediate



Op	Funct$_5$	Funct$_0$	Type	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp
01	X	0	STR	0	X	1	1	01	0	10	0

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$_5$ (Ibar) = X: don't care

funct$_0$ (L) = 0: not a LDR

Branch = 0: no branch

MemtoReg = X: don't care

ALUSrc = 1: Imm, no 2nd reg

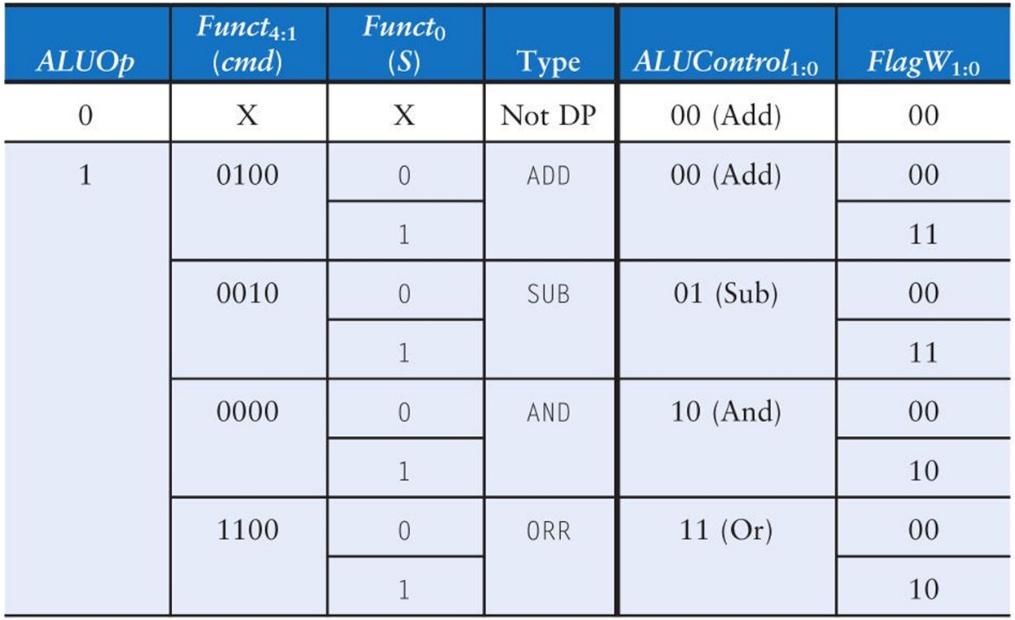
ImmSrc = 01: 12-bit zero extended

RegSrc = 01: 19:16 (RA1) and 15:12 (RA2)

ALU Decoder Truth Table

The ALUControl asserts FlagW to update the status flags when the S-bit is set.

`add`,`sub` update all flags (11), whereas `and`,`orr` only update the N and Z flags. (10)



ALUOp	*Funct*$_{4:1}$ (*cmd*)	*Funct*$_0$ (*S*)	Type	*ALUControl*$_{1:0}$	*FlagW*$_{1:0}$
0	X	X	Not DP	00 (Add)	00
1	0100	0	ADD	00 (Add)	00
		1			11
	0010	0	SUB	01 (Sub)	00
		1			11
	0000	0	AND	10 (And)	00
		1			10
	1100	0	ORR	11 (Or)	00
		1			10

PC Logic

The PC Logic checks if the instruction is a write to `r15` or a branch such that the PC should be updated

```
PCS = ((Rd == 15) && RegW) || Branch
```

PCS may be killed by the conditional logic before it is sent to the datapath as PCSrc.

Conditional Logic

- The conditional logic determines whether the instruction should be executed (CondEx) based on the cond field and the current values of the N, Z, C, and V flags (Flags3:0).
- If the instruction should not be executed, the write enables and PCSrc are forced to 0 so that the instruction does not change the architectural state.
- The conditional logic also updates some or all of the flags from the ALUFlags when FlagW is asserted by the ALU Decoder and the instruction's condition is satisfied (CondEx =1).

Summary

We have discussed the following topics

- Single Cycle Microarchitecture
- Datapaths for Memory, Data Processing and Branching examples
- Control unit
  - Multiplexers
  - Truth Tables and decoder Logic
  - Conditional Logic
-