CISC260 Machine Organization and Assembly Language

ARM and Its ISA

Credits: Some of the slides are adopted/adapted from the Harris&Harris

[quiz] For any logic devise, its output is solely determined by the input. Correct or not?

A. Yes
B. No
C. Depends
D. Don't know

ARM (32bit, single cycle computer)

- Data path
- Parse/Decoder
- Control/Mux

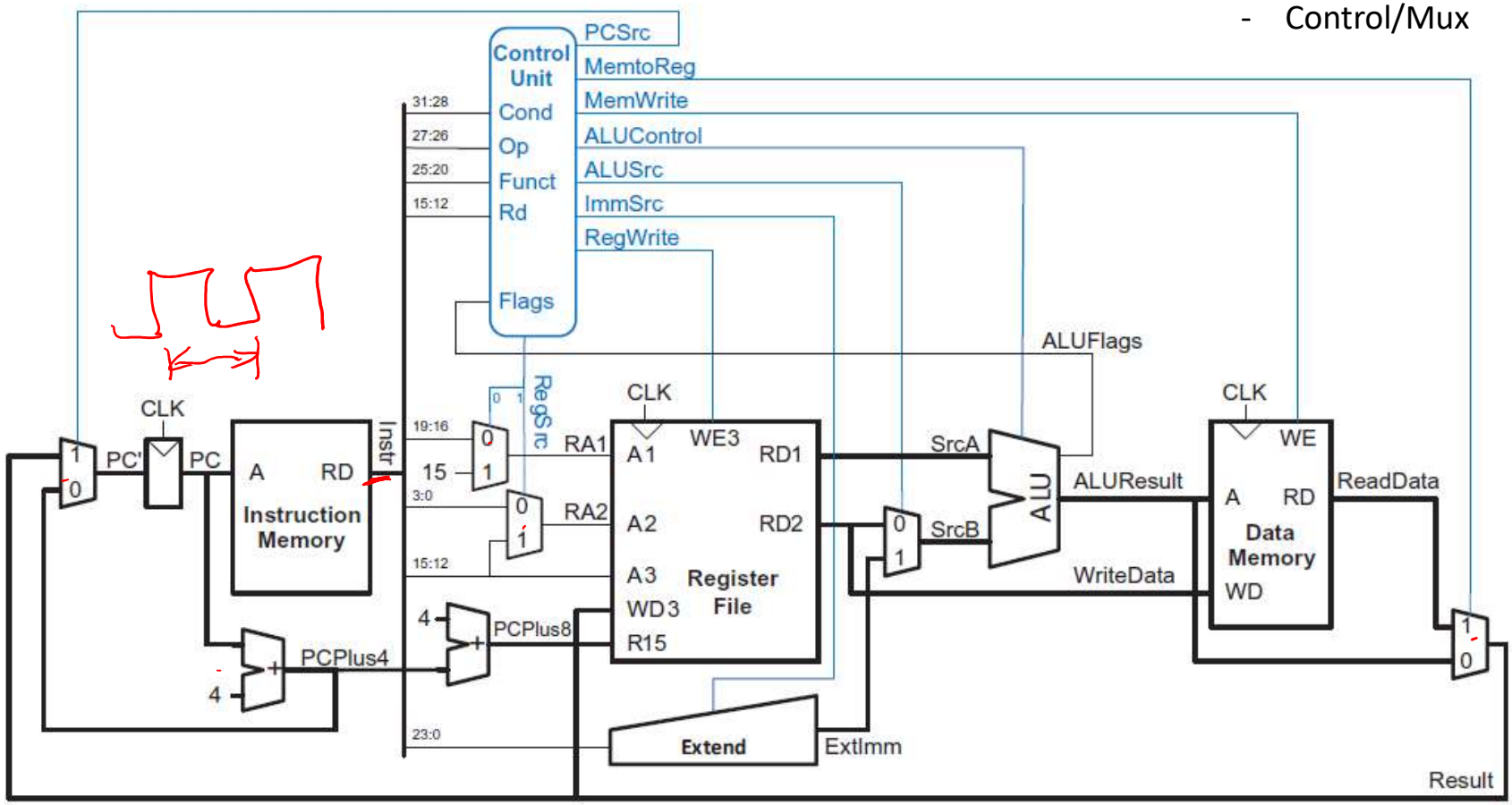


Figure 7.13 Complete single-cycle processor

ARM (32bit, multiple cycle pipelined computer)

Stages:
- Instruction Fetch
- Instruction Decode
- Execution
- Memory
- Write Back

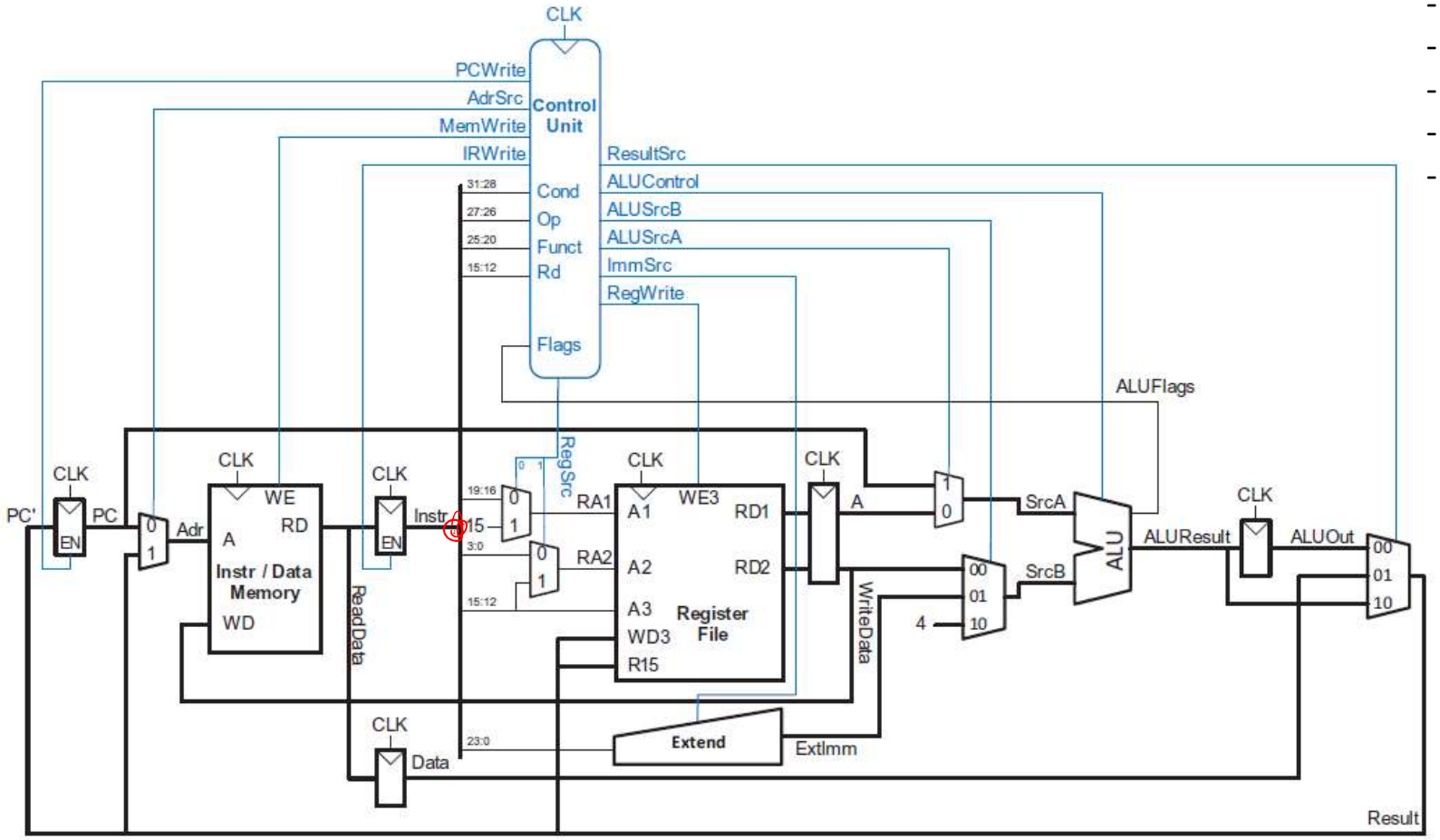


Figure 7.30 Complete multicycle processor

A Programmer's Perspective

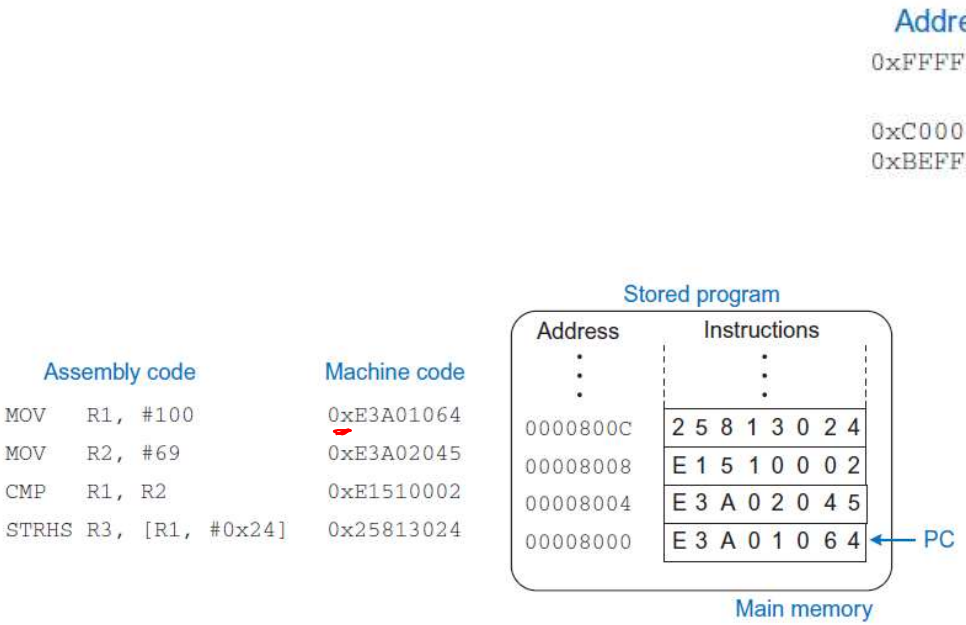


Figure 6.28 Stored program

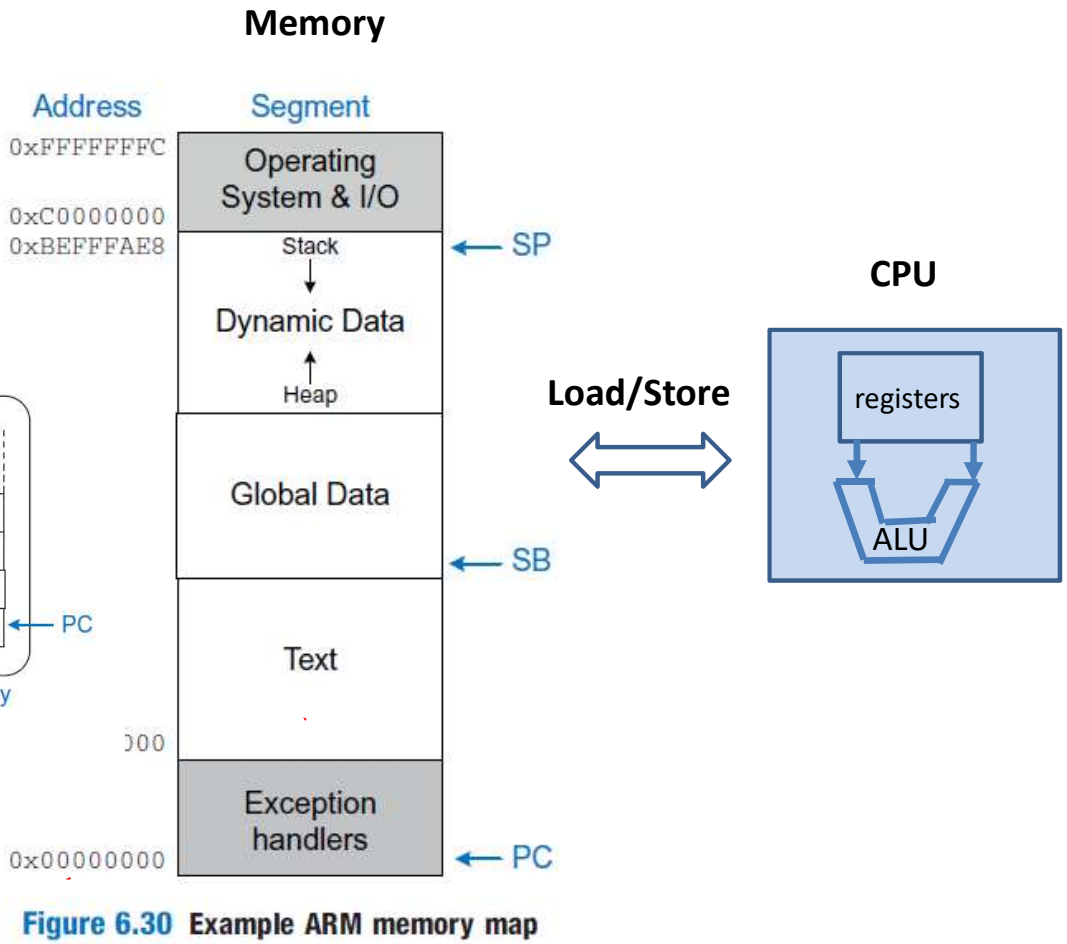
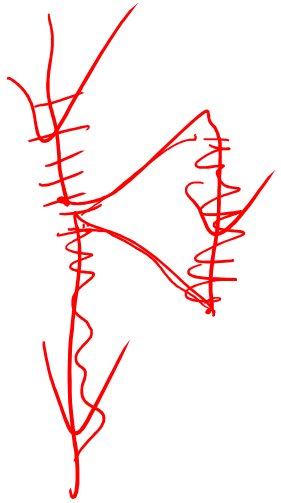


Figure 6.30 Example ARM memory map

Table 6.1 ARM register set

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



Features of ARM Instruction Set

Common to RISC:

- Load-Store architecture
- All instructions are the same length (32-bit)
- 3-address instruction

Unique to ARM

- Conditional execution of every instruction
- Possible to load/store multiple registers at once
- Possible to combine shift and ALU operations in a single instruction

Basic ARM Instructions

Data processing:		Meaning
ADD	r1, r2, r3	r1 ← r2 + r3
SUB	r1, r2, r3	r1 ← r2 - r3
AND	r1, r2, r3	r1 ← r2 & r3
ORR	r1, r2, r3	r1 ← r2 | r3
MVN	r1, r2	r1 ← ~ r2
LSL	r1, r2, #10	r1 ← r2 <<10
LSR	r1, r2, #10	r1 ← r2 >>10

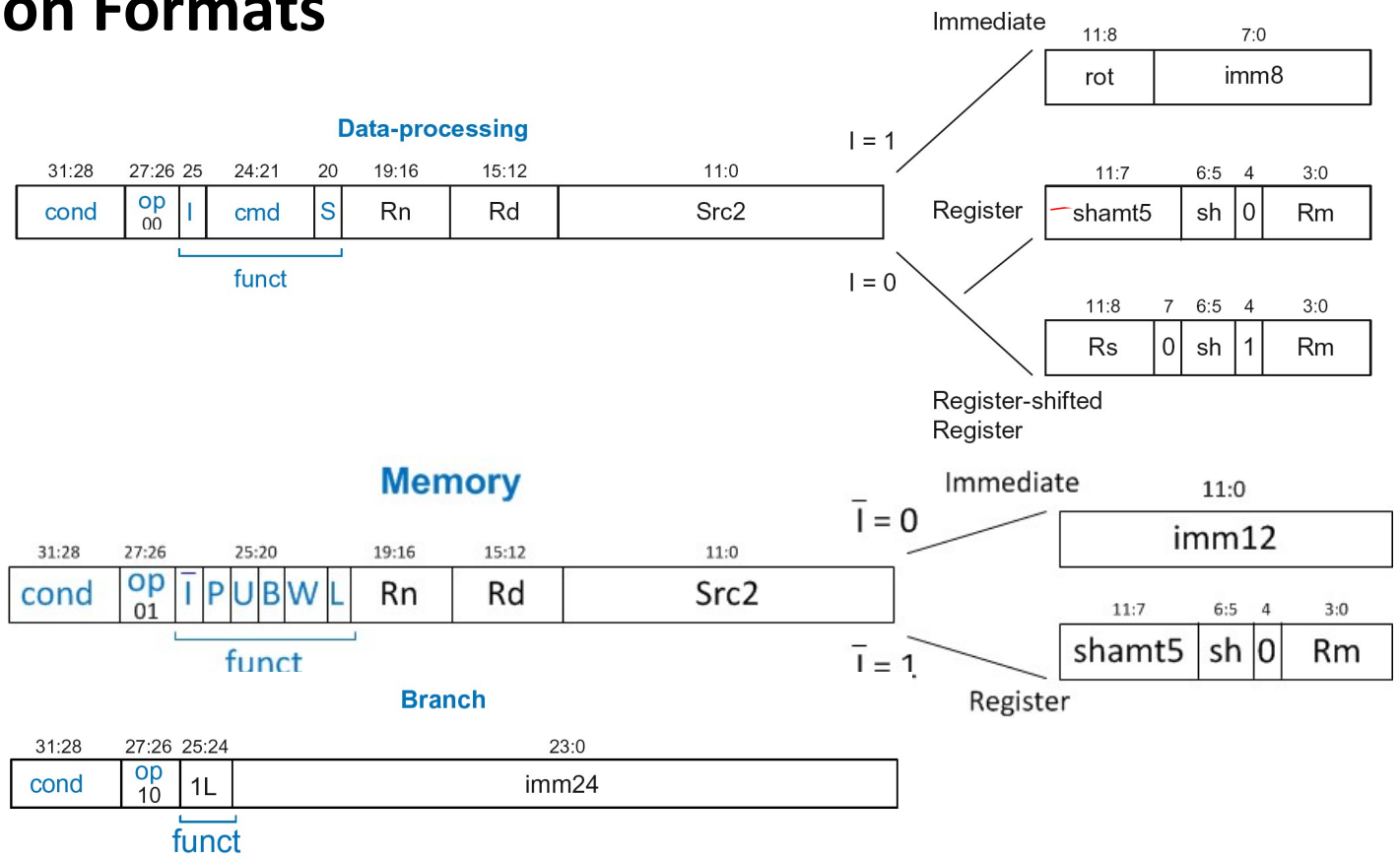
Data Transfer (Memory):

LDR	r1, [r2, #20]	r1 ← Memory[r2 + 20]
STR	r1, [r2, #20]	Memory[r2 + 20] ← r1
SWAP	r1, [r2, #20]	r1 ↔ Memory [r2+20]
MOV	r1, r2	r1 ← r2

Branching:

CMP	r1, r2	Condition flag NZCV is set per the outcome of r1-r2
B	label	PC ← Address of label
BL	label	PC ← Address of label, r14 ← PC+4

Instruction Formats

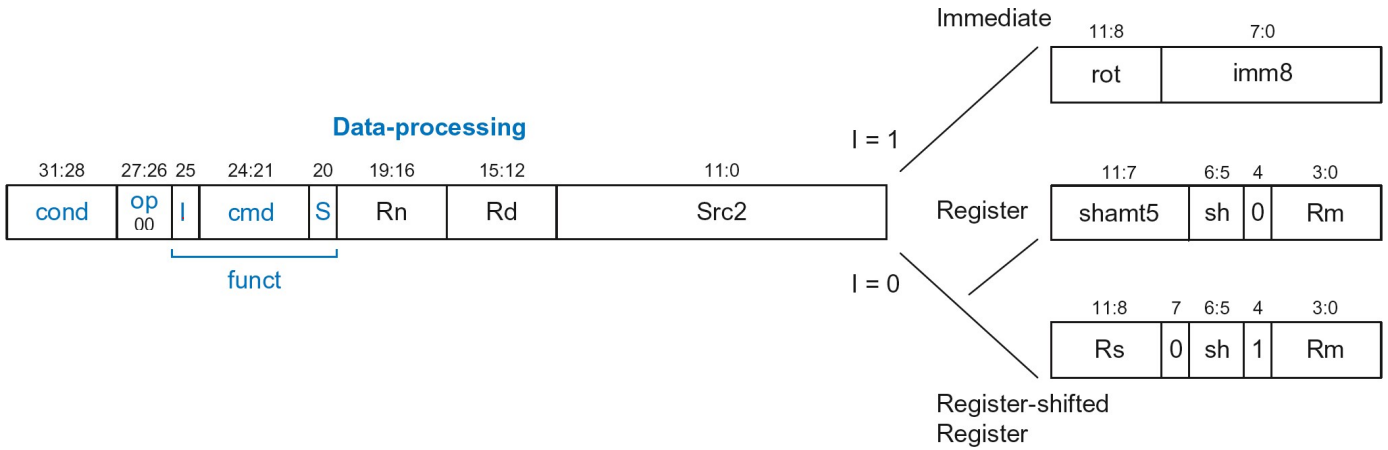


These formats allow variations to the basic instructions:

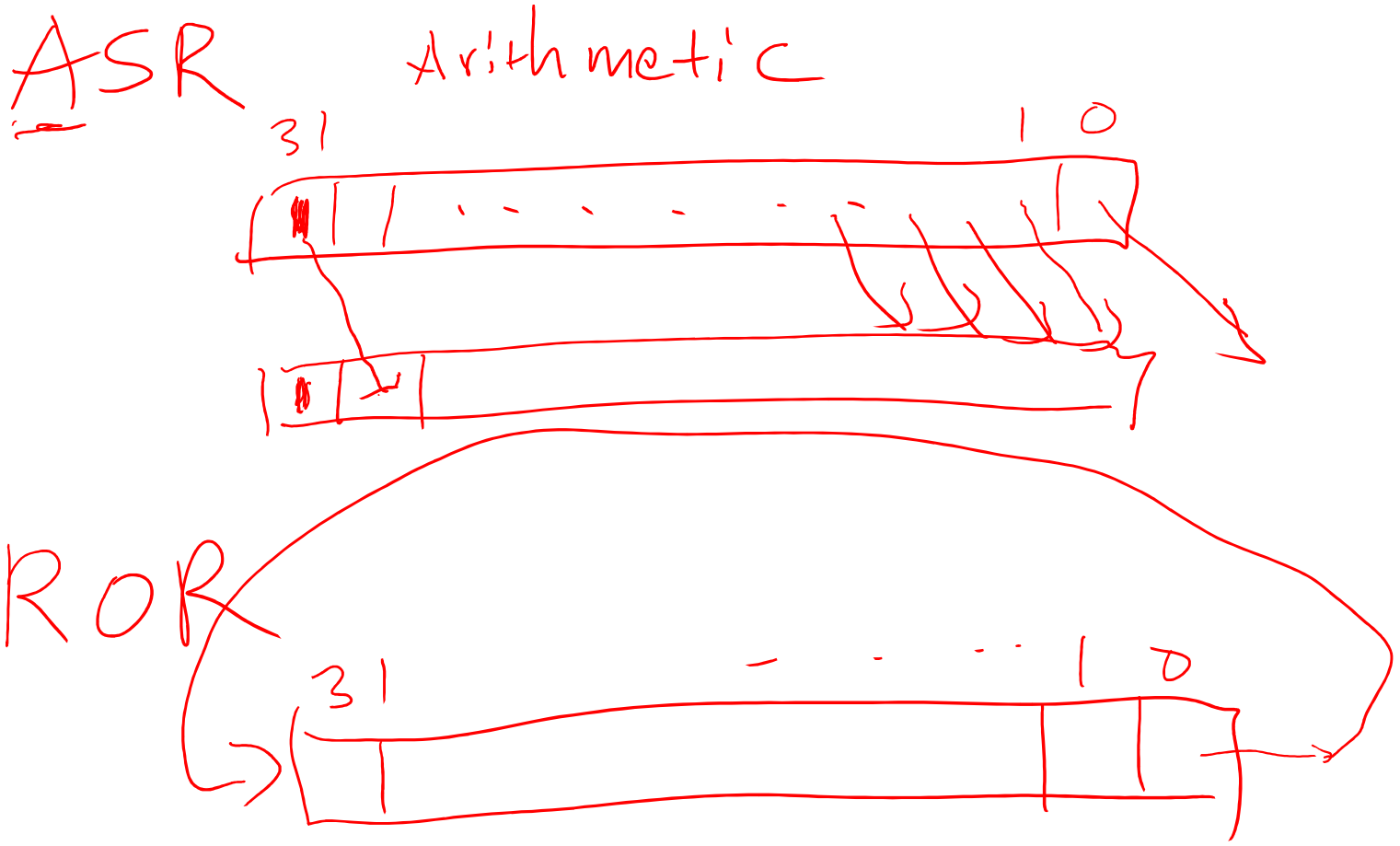
- **operand types, e.g., ADD r1, r2, #4**
- condition suffix, e.g., BEQ, ADDEQ , ADDS

Variation on the operands for data processing instructions

```
ADD   r0, r1, r2
ADD   r0, r1, #4              ; r0 = r1 + 4
ADD   r0, r1, r2, LSL #2      ; r0 = r1 + (r2<<2)
ADD   r0, r1, r2, LSL r3      ; r0 = r1 + r2 x 2^r3
```



Shift Type	*sh*
LSL	00_2
LSR	01_2
ASR	10_2
ROR	11_2


ASR
Arithmetic
31
1
0
ROR
31
1
0

Immediate value from Src2

- Src2 has 12 bits and can encode unsigned integers 0 – 4095
- But ARM uses a different encoding

    32-bit immediate value = imm8 ROR (rot x 2)

    imm8: 0 – 255

    rot x 2: 2, 4, 6, …, 30

    Available immediate values:

        0 – 255,

        256, 260, 264, …, 1020 (ror 30)

        …

Note: As programmer, you put in an immediate value and the assembler will let you know if that value can be encoded.

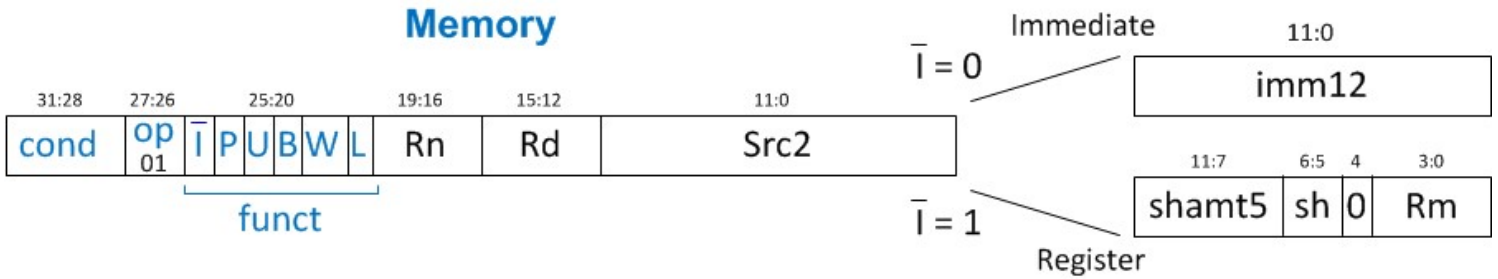
Format for data transfer instructions

- ***funct***:
  - ***I***: Immediate bar
  - ***P***: Preindex
  - ***U***: Add
  - ***B***: Byte
  - ***W***: Writeback
  - ***L***: Load

Value	*I*	*U*
0	**Immediate** offset in *Src2*	**Subtract** offset from base
1	**Register** offset in *Src2*	**Add** offset to base

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

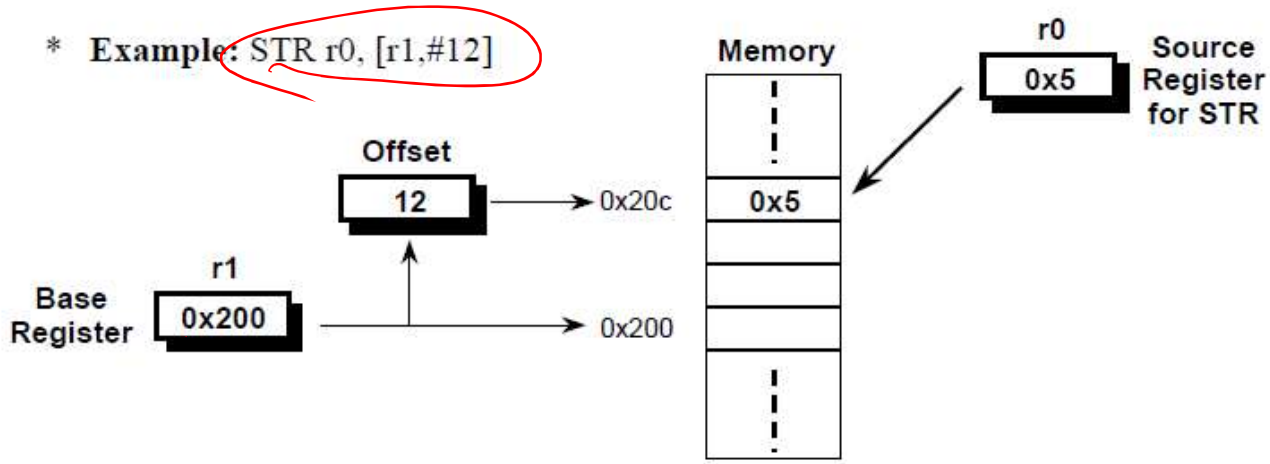
P	*W*	Indexing Mode
0	1	Not supported
0	0	Postindex
1	0	Offset
1	1	Preindex



Variation on the operands for data transferring instructions

```
LDR    r0, [ r1]              ; r0 = M[r1]
LDR    r0, [r1, #4]           ; r0 = M[r1 + 4]
LDR    r0, [r1, r2]           ; r0 = M[r1 + r2]
LDR    r0, [r1, r2, LSL #2]   ; r0 = M[r1 + r2<<2]
LDR    r0, [r1, #4]!          ; r0 = M[r1 + 4], r1 = r1 + 4
LDR    r0, [r1], #4           ; r0 = M[r1], r1 = r1 + 4
LDR    r0, [r1, r2]!          ; r0 = M[r1 + r2], r1 = r1 + r2
LDR    r0, [r1], r2           ; r0 = M[r1], r1 = r1 + r2
```

LDR, LDRH, LDRB

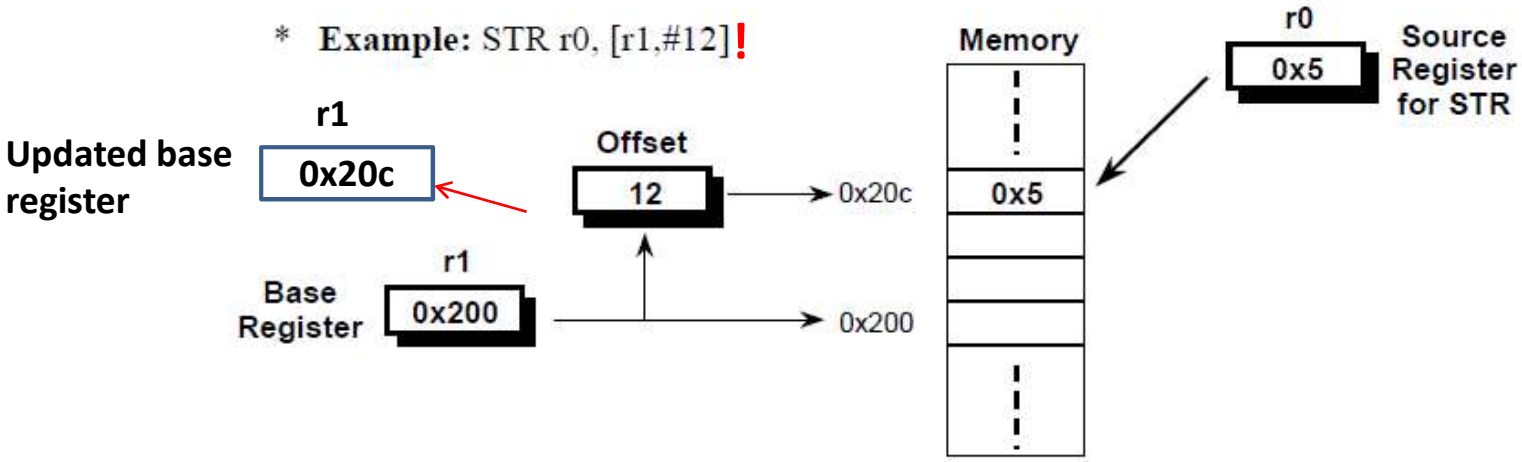


STR r0, [r1, r2, LSL #2] **@ address = r1 + 4 x r2**
@ if r2 has value 3, this has the same effect of STR r0, [r1,#12].

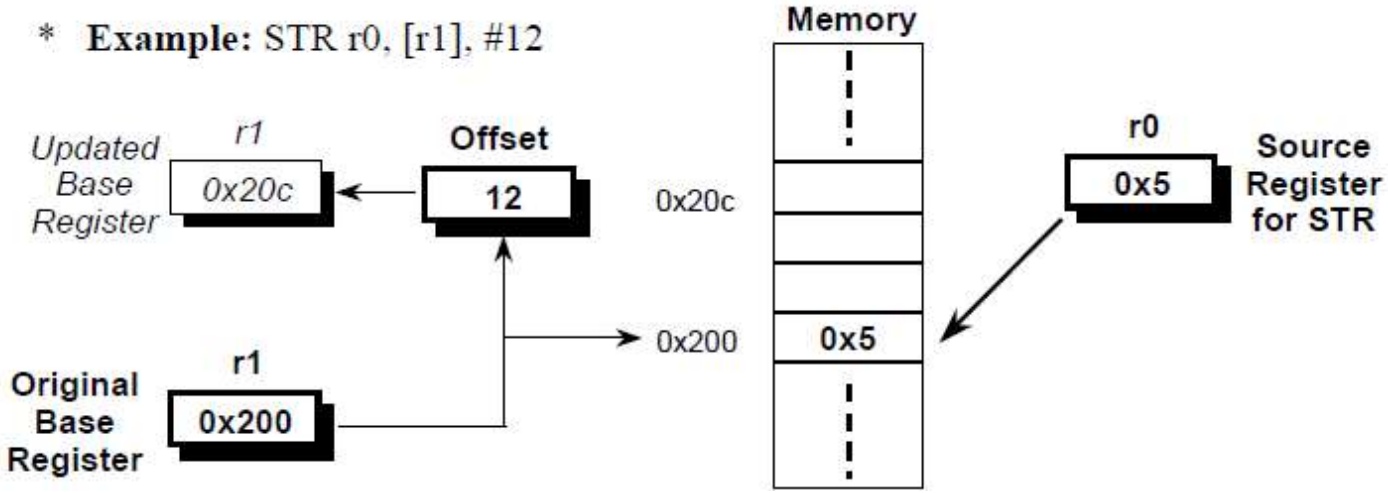
STR r0, [r1, #12]! **@ pre-indexing**

STR r0 [r1], #12 **@ post-indexing**

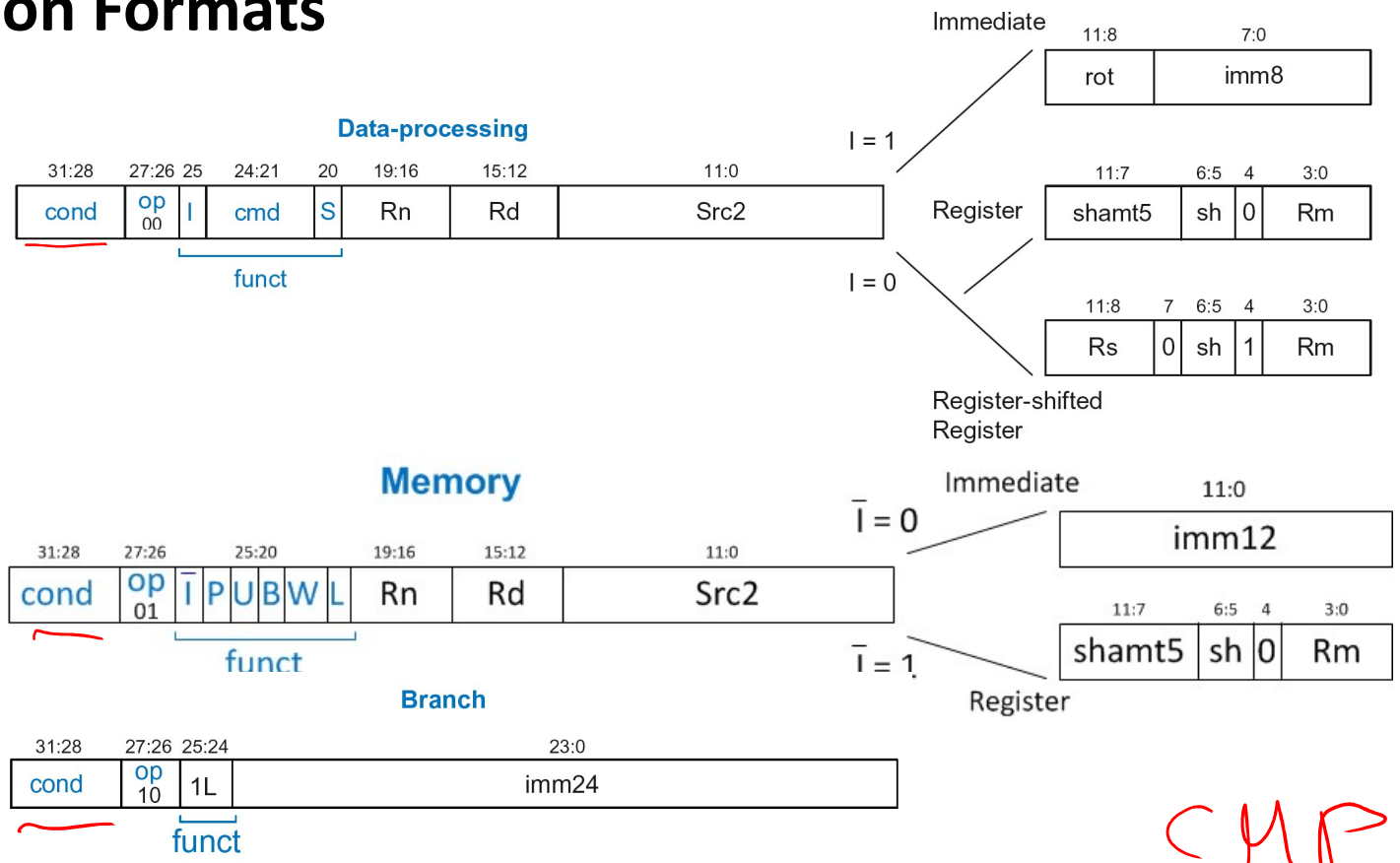
Pre-indexing



Post-indexing



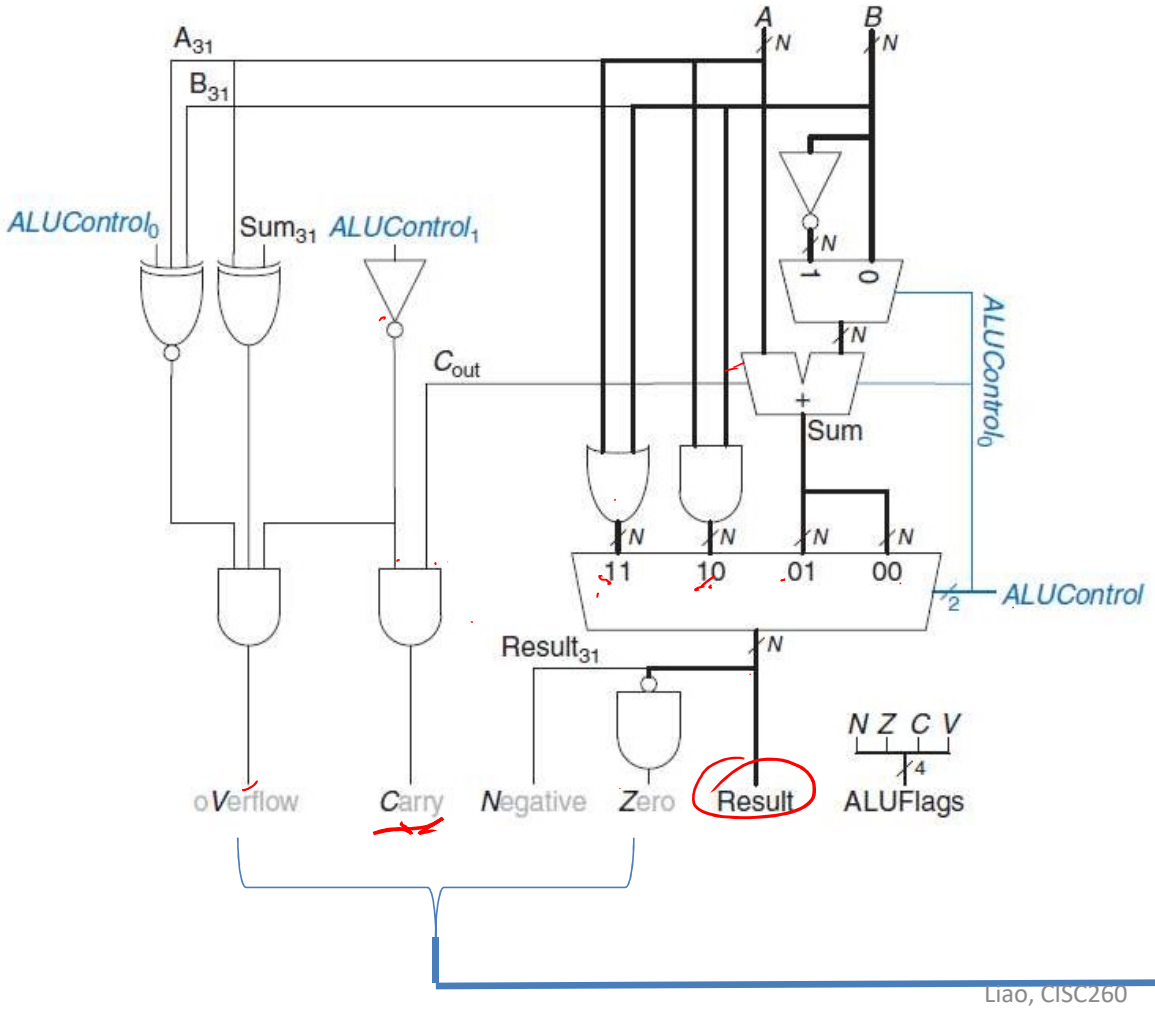
Instruction Formats



These formats allow variations to the basic instructions:

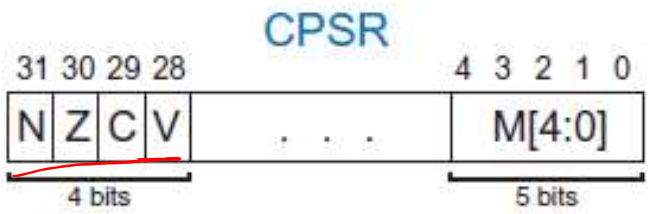
- operand types, e.g., ADD r1, r2, #4
- **condition suffix, e.g., BEQ, ADDEQ , ADDS**

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



Condition flags

- can be set by an instruction, e.g., CMP, ADDS
- can be used to conditionally execute an instruction, e.g., BEQ, ADDEQ



Setting the Condition Flags: *NZCV*

- **Method 1:** Compare instruction: `CMP`

  Example: `CMP R5, R6`

  - Performs: R5-R6
  - Sets flags: If result is 0 (Z=1), negative (N=1), etc.
  - Does not save result

- **Method 2:** Append instruction mnemonic with **`S`**

  Example: `ADD`**`S`** `R1, R2, R3`

  - Performs: R2 + R3
  - Sets flags: If result is 0 (Z=1), negative (N=1), etc.
  - Saves result in R1

- Instruction may be ***conditionally executed*** based on the condition flags
- Condition of execution is encoded as a ***condition mnemonic*** appended to the instruction mnemonic

  Example:
  ```
  CMP    R1, R2
  SUBNE  R3, R5, R8
  ```

  - **NE:** condition mnemonic
  - `SUB` will only execute if R1 ≠ R2 (i.e., Z = 0)

Example:

```
CMP   R5, R9          ; performs R5-R9
                      ; sets condition flags

SUBEQ R1, R2, R3      ; executes if R5==R9 (Z=1)
ORRMI R4, R0, R9      ; executes if R5-R9 is
                      ; negative (N=1)
```

Suppose R5 = 17, R9 = 23:

CMP performs: 17 – 23 = -6 (Sets flags: *N*=1, *Z*=0, *C*=0, *V*=0)

SUBEQ **doesn't execute** (they aren't equal: *Z*=0)

ORRMI **executes** because the result was negative (*N*=1)

Signed versus unsigned comparison

Suppose

r0 = 1111 1111 1111 1111 1111 1111 1111 1111_{two}
r1 = 0000 0000 0000 0000 0000 0000 0000 0001_{two}

If the following instructions are executed

0x0000 1000: CMP r0, r1
0x0000 1004: BLO L1
0x0000 1008: BLT L2

r0 − r1

Where will be the PC at?

A: L1
B: L2
C: 0x0000 100C
D: 0x0000 1010

Value	Meaning	Value	Meaning
0	EQ (EQual)	8	HI (unsigned HIgher)
1	NE (Not Equal)	9	LS (unsigned Lower or Same)
2	HS (unsigned Higher or Same)	10	GE (signed Greater than or Equal)
3	LO (unsigned LOwer)	11	LT (signed Less Than)
4	MI (MInus, <0)	12	GT (signed Greater Than)
5	PL - (PLus, >=0)	13	LE (signed Less Than or Equal)
6	VS (oVerflow Set, overflow)	14	AL (Always)
7	VC (oVerflow Clear, no overflow)	15	NV (reserved)

r0 = 1111 1111 1111 1111 1111 1111 1111 1111
r1 = 0000 0000 0000 0000 0000 0000 0000 0001

So, the result of r0 - r1 is computed as r0 + (-r1) = r0 + (~r1 +1)

```
    1111 1111 1111 1111 1111 1111 1111 1111    (r0)
+)  1111 1111 1111 1111 1111 1111 1111 1111    (~r1 + 1)
-------------------------------------------------------------------------------
  11111 1111 1111 1111 1111 1111 1111 1110
```

Based on this result,
N = 1 (the result is negative, treated as two's complement)
Z = 0 (the result is not zero)
C = 1 (there is carry out of the left-most bit)
V = 0 (there is no overflow)

Therefore, the instruction "BLO" is not taken because of suffix "LO" indicates unsigned lower, which is only taken when the carry bit is clear. – because if r0 is unsigned lower than r1, r0 – r1 will never have a carry.

Instead instruction "BLT" is taken when N != V, which is the case as shown above.

B.3 BRANCH INSTRUCTIONS

Figure B.4 shows the encoding for branch instructions (B and BL) and Table B.4 describes their operation.

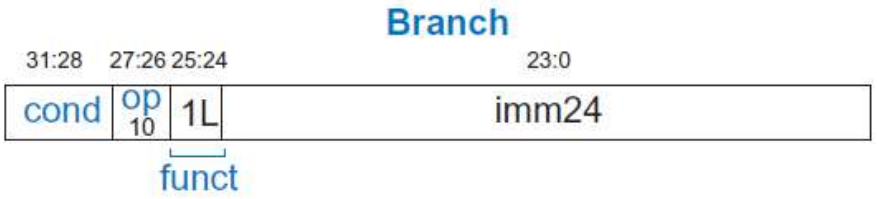


Figure B.4 Branch instruction encoding

Table B.4 Branch instructions

L	Name	Description	Operation
0	B label	Branch	PC ← (PC+8)+imm24 << 2
1	BL label	Branch with Link	LR ← (PC+8) - 4; PC ← (PC+8)+imm24 << 2

PC-relative addressing: Imm24 = # of words that the branch label is away from PC+8

ARM (32bit, multiple cycle pipelined computer)

Stages:
- Instruction Fetch
- Instruction Decode
- Execution
- Memory
- Write Back

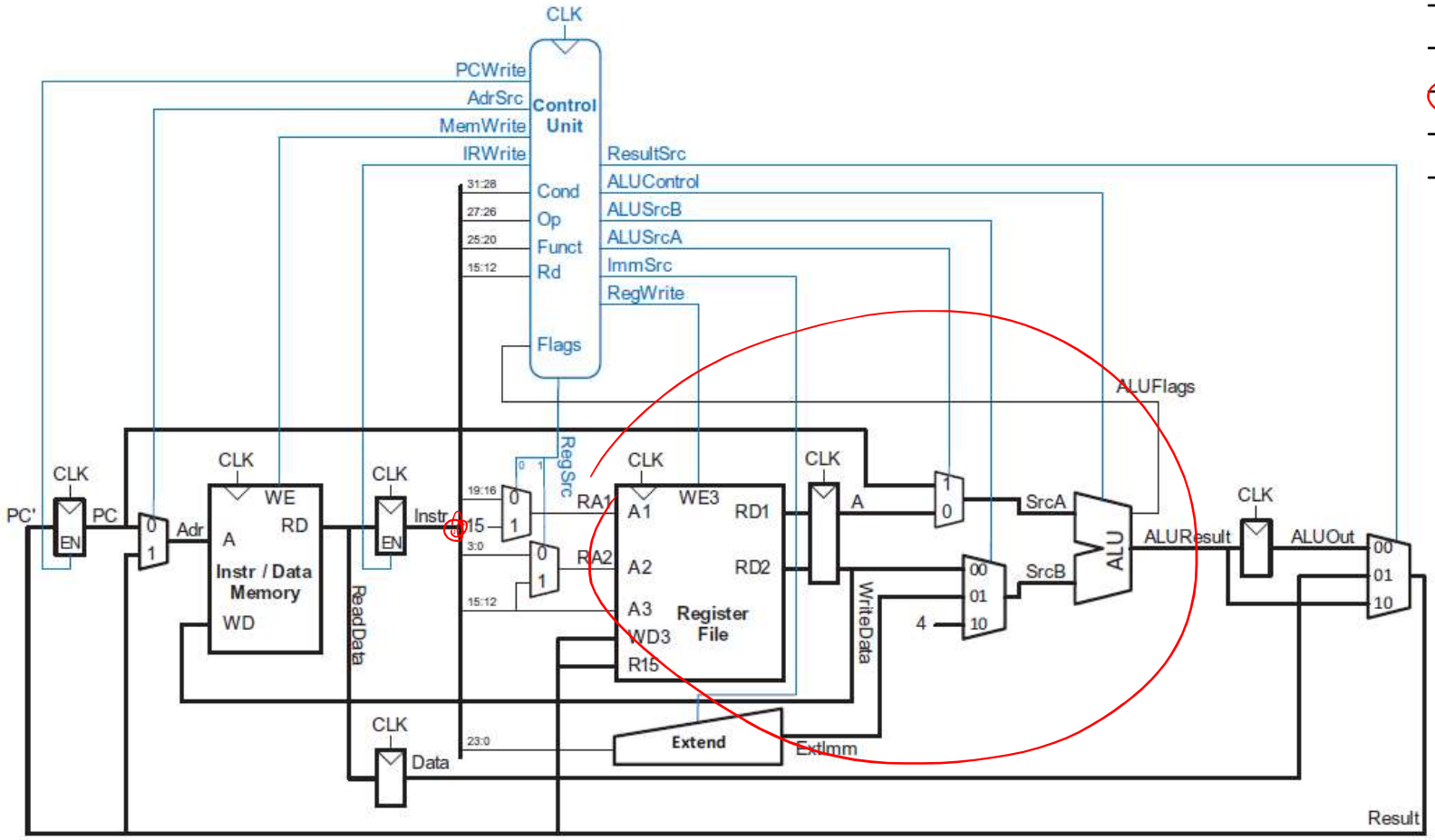


Figure 7.30 Complete multicycle processor

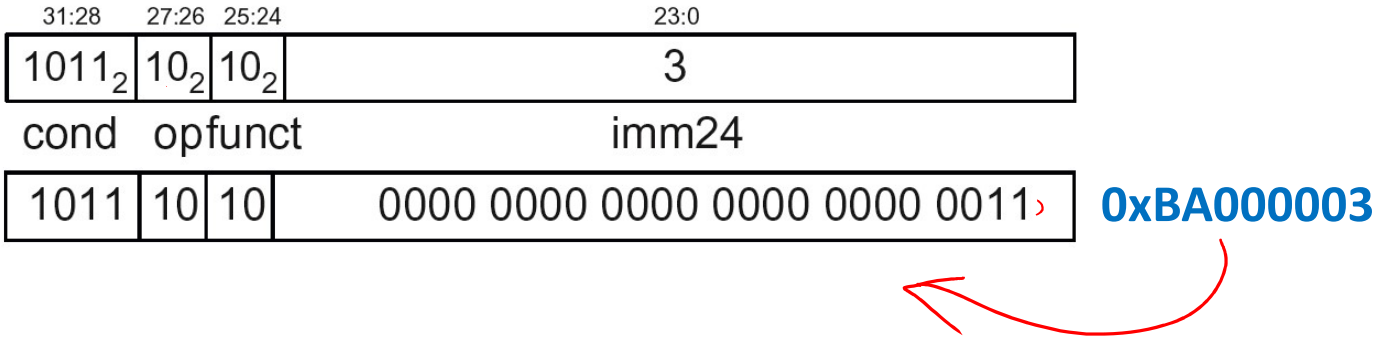
ARM assembly code

```
0xA0          BLT  THERE        ← PC
0xA4          ADD  R0, R1, R2
0xA8          SUB  R0, R0, R9   ← PC+8
0xAC          ADD  SP, SP, #8
0xB0          MOV  PC, LR
0xB4  THERE   SUB  R0, R0, #1   ← BTA
0xB8          BL   TEST
```

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

BTA = (PC + 8) + signext(imm24 << 2)

Field Values



ARM assembly code

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

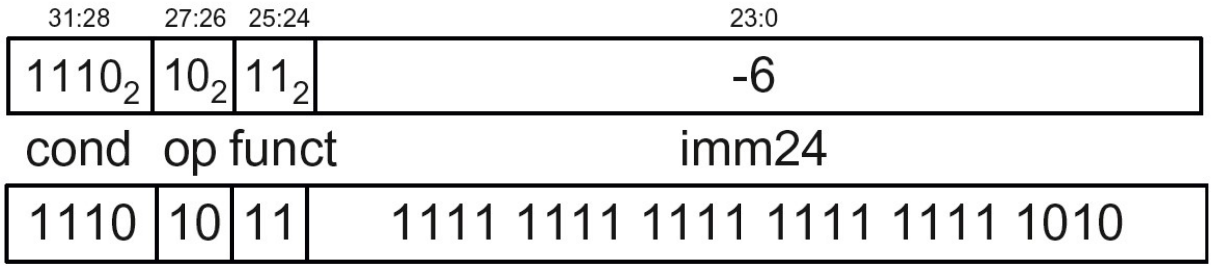
= (PC + 8) + signext(imm24 << 2)

- PC = 0x8050
- PC + 8 = 0x8058
- `TEST` label is 6 instructions before PC+8
- So, *imm24* = -6

Maximum 2^{23} instructions backwards

Maximum $2^{23}-1$ instructions forwards

Field Values



0xEBFFFFFA

Table 6.12 ARM operand addressing modes

Operand Addressing Mode	Example	Description
Register		
Register-only	ADD R3, R2, R1	R3 ← R2 + R1
Immediate-shifted register	SUB R4, R5, R9, LSR #2	R4 ← R5 − (R9 >> 2)
Register-shifted register	ORR R0, R10, R2, ROR R7	R0 ← R10 | (R2 ROR R7)
Immediate	SUB R3, R2, #25	R3 ← R2 − 25
Base		
Immediate offset	STR R6, [R11, #77]	mem[R11+77] ← R6
Register offset	LDR R12, [R1, −R5]	R12 ← mem[R1 − R5]
Immediate-shifted register offset	LDR R8, [R9, R2, LSL #2]	R8 ← mem[R9 + (R2 << 2)]
PC-Relative	B LABEL1	Branch to LABEL1

PC ←PC+8 +(offset)

Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary
0_{hex}	0000_{two}	4_{hex}	0100_{two}	8_{hex}	1000_{two}	c_{hex}	1100_{two}
1_{hex}	0001_{two}	5_{hex}	0101_{two}	9_{hex}	1001_{two}	d_{hex}	1101_{two}
2_{hex}	0010_{two}	6_{hex}	0110_{two}	a_{hex}	1010_{two}	e_{hex}	1110_{two}
3_{hex}	0011_{two}	7_{hex}	0111_{two}	b_{hex}	1011_{two}	f_{hex}	1111_{two}

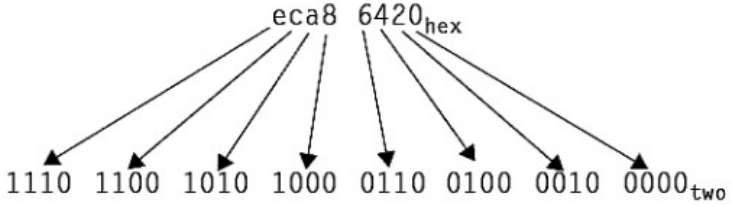
Binary-to-Hexadecimal and Back

Convert the following hexadecimal and binary numbers into the other base:

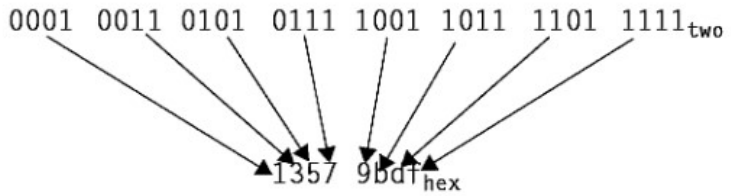
$\texttt{eca8 6420}_{hex}$

$0001\ 0011\ 0101\ 0111\ 1001\ 1011\ 1101\ 1111_{two}$

Just a table lookup one way:



And then the other direction too:



E0813002 : add r3, r1, r2

1110 0000 1000 0001 0011 0000 0000 0010

cond F I opcode S rn rd

C0813002 : addgt r3, r1, r2

1100

E2813002 : add r3, r1, #2

1110 0010 1000 0001 0011 0000 0000 0010

I S rn r

EAFFFFFB : b -12

1110 1010 1111 1111 1111 1111 1111 0101

24-bit

a. Sign-ext 30 1011 00 = 20

1111 1111 1111 1111 1111 1111 1110 1100 = -20

+) old PC + 8

PC = old PC - 12