UNIVERSITY OF TEXAS ARLINGTON

Computer Organization & Assembly Language Programming (CSE 2312)

Lecture 8: Instructions Review and Addressing Modes

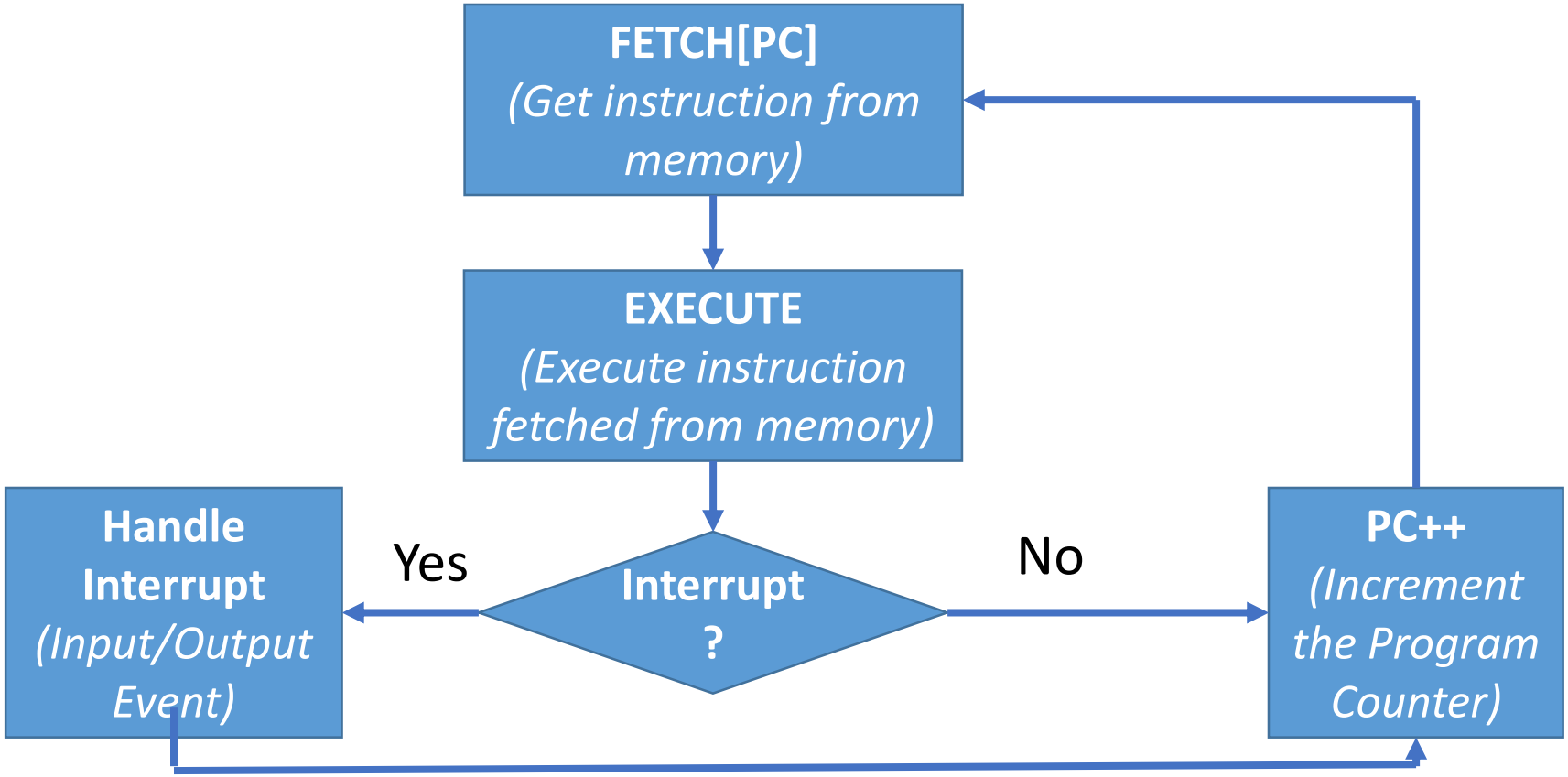
Taylor Johnson

Announcements and Outline

- Quiz 3 on Blackboard site (due 11:59PM Friday)
  - Review binary arithmetic and Boolean operations
- Homework 2 due today
- Homework 3 assigned today
  - Finish reading chapter 2 (ARM version on Blackboard site)

- Instructions Review
- Addressing Modes

Review: Abstract Processor Execution Cycle



Review: Memory Cells and Addresses

- ***Memory cell***: a piece of memory that contains a specific number of bits
  - How many bits depends on the architecture
  - In modern architectures, it is almost universal that a cell contains *8 bits (1 byte)*, and that will be also our convention in this course
- ***Memory address***: a number specifying a location of a memory cell containing data
  - Essentially, a number specifying the location of a byte of memory

Review: Operands Types

- Register operand: operand comes from the binary valued stored in a particular register in the CPU
  - Example: `add r0, r1, r2`
  - C code: `r0 = r1 + r2;`
- Immediate operand: operand value comes from instruction itself
  - Example: `add r0, r1, #1`
  - C code: `r0 = r1 + 1;`
- Memory operand: operand refers to memory
  - Example: `str r0, [r1]`
  - C code (roughly): `MEM[r1] = r0;`
  - Only for load / store instructions!
  - Several addressing modes (more on this later)

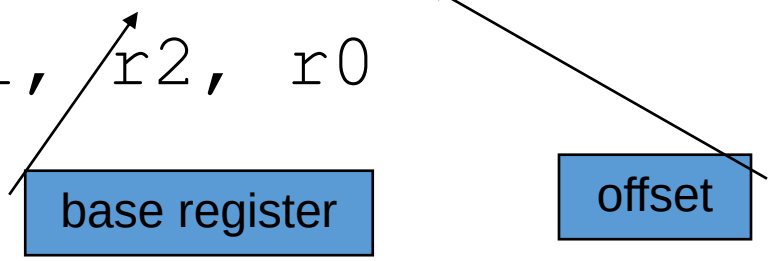
Review: Memory Operand Example 1

- C code:

```
g = h + A[8];
```

  - g in r1, h in r2, base address of A in r3
- Compiled ARM code:
  - Index 8 requires offset of 8 words
    - 4 bytes per word

```
@ load word
ldr r0, [r3, #32] @ r0 = MEM[r3 + 32]
add r1, r2, r0
```



Review: Memory Operand Example 2

- C code:

```
A[12] = h + A[8];
```

  - h in r2, base address of A in r3

- Compiled ARM code:
  - Index 8 requires offset of 32 (8 bytes, 4 bytes per word)

```
@ load word
ldr r0, [r3,#32] @ r0 = MEM[r3 + 32]
add r0, r2, r0
@ store word
str r0, [r3, #48] @ MEM[r3 + 48] = r0
```

Review: Immediate Operands

- Constant data specified in an instruction

```
add r3, r3, #4
```

- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

ARM Instruction Formats

31 28	27 26	25	24 21	20	19 16	15 12	11 8	7 6	5	4	3 0	Instruction type
Cond	0 0	I	Opcode	S	Rn	Rd	Operand2					Data processing / PSR Transfer
Cond	0 0 0 0 0 0		A	S	Rd	Rn	RS	1 0 0 1			Rm	Multiply
Cond	0 0 0 0 1		U	A	S	RdHi	RdLo	RS	1 0 0 1		Rm	Long Multiply
Cond	0 0 0 1 0		B	0 0		Rn	Rd	0 0 0 0	1 0 0 1		Rm	Swap
Cond	0 1	I	P	U	B	W	L	Rn	Rd	Offset		Load/Store Byte/Word
Cond	1 0 0		P	U	S	W	L	Rn	Register List			Load/Store Multiple
Cond	0 0 0		P	U	1	W	L	Rn	Rd	Offset1	1	S	H	1	Offset2	Halfword transfer: Immediate offset
Cond	0 0 0		P	U	0	W	L	Rn	Rd	0 0 0 0	1	S	H	1	Rm	Halfword transfer: Register offset
Cond	1 0 1		L	Offset												Branch
Cond	0 0 0 1		0 0 1 0			1 1 1 1		1 1 1 1	1 1 1 1	0 0 0 1		Rn				Branch Exchange
Cond	1 1 0	P	U	N	W	L	Rn	CRd	CPNum	Offset						Coprocessor data transfer
Cond	1 1 1 0		Op1				CRn	CRd	CPNum	Op2	0	CRm				Coprocessor data operation
Cond	1 1 1 0		Op1		L		CRn	Rd	CPNum	Op2	1	CRm				Coprocessor register transfer
Cond	1 1 1 1		SWI Number													Software interrupt

ARM Instruction Formats

Name	Format	Example								Comments
ADD	DP	14	0	0	4	0	2	1	3	ADD r1,r2,r3
SUB	DP	14	0	0	2	0	2	1	3	SUB r1,r2,r3
LDR	DT	14	1	24			2	1	100	LDR r1,100(r2)
STR	DT	14	1	25			2	1	100	STR r1,100(r2)
Field size		4 bits	2 bits	1 bit	4 bits	1 bit	4 bits	4 bits	12 bits	All ARM instructions are 32 bits long
DP format	DP	Cond	F	I	Opcode	S	Rn	Rd	Operand2	Arithmetic instruction format
DT format	DT	Cond	F	Opcode			Rn	Rd	Offset12	Data transfer format

- DP: data processing instructions: transform data (arithmetic, etc.)
- DT: data transfer instructions: move data around (load from memory, store to memory, etc.)

Name	Format	Example								Comments
Field size		4 bits	2 bits	1 bit	4 bits	1 bit	4 bits	4 bits	12 bits	All ARM instructions are 32 bits long
DP format	DP	Cond	F	I	Opcode	S	Rn	Rd	Operand2	Arithmetic instruction format
DT format	DT	Cond	F	Opcode			Rn	Rd	Offset12	Data transfer format
Field size		4 bits	2 bits	2 bits	24 bits					
BR format	BR	Cond	F	Opcode	signed_immed_24					B and BL instructions

ARM Instructions

Loads

LDRSB DST,ADDR	Load signed byte (8 bits)
LDRB DST,ADDR	Load unsigned byte (8 bits)
LDRSH DST,ADDR	Load signed halfwords (16 bits)
LDRH DST,ADDR	Load unsigned halfwords (16 bits)
LDR DST,ADDR	Load word (32 bits)
LDM S1,REGLIST	Load multiple words

Stores

STRB DST,ADDR	Store byte (8 bits)
STRH DST,ADDR	Store halfword (16 bits)
STR DST,ADDR	Store word (32 bits)
STM SRC,REGLIST	Store multiple words

Shifts/rotates

LSL DST,S1,S2IMM	Logical shift left
LSR DST,S1,S2IMM	Logical shift right
ASR DST,S1,S2IMM	Arithmetic shift right
ROR DSR,S1,S2IMM	Rotate right

Boolean

TST DST,S1,S2IMM	Test bits
TEQ DST,S1,S2IMM	Test equivalence
AND DST,S1,S2IMM	Boolean AND
EOR DST,S1,S2IMM	Boolean Exclusive-OR
ORR DST,S1,S2IMM	Boolean OR
BIC DST,S1,S2IMM	Bit clear

S1 = source register
S2IMM = source register or immediate
S3 = source register (when 3 are used)
DST = destination register
D1 = destination register (1 of 2)
D2 = destination register (2 of 2)

ADDR = memory address
IMM = immediate value
REGLIST = list of registers
PSR = processor status register
cc = branch condition

ARM Instructions

Arithmetic

ADD DST,S1,S2IMM	Add
ADD DST,S1,S2IMM	Add with carry
SUB DST,S1,S2IMM	Subtract
SUB DST,S1,S2IMM	Subtract with carry
RSB DST,S1,S2IMM	Reverse subtract
RSC DST,S1,S2IMM	Reverse subtract with carry
MUL DST,S1,S2	Multiply
MLA DST,S1,S2,S3	Multiple and accumulate
UMULL D1,D2,S1,S2	Unsigned long multiple
SMULL D1,D2,S1,S2	Signed long multiple
UMLAL D1,D2,S1,S2	Unsigned long MLA
SMLAL D1,D2,S1,S2	Signed long MLA
CMP S1,S2IMM	Compare and set PSR

Transfer of control

Bcc IMM	Branch to PC+IMM
BLcc IMM	Branch with link to PC+IMM
BLcc S1	Branch with link to reg add

Miscellaneous

MOV DST,S1	Move register
MOVT DST,IMM	Move imm to upper bits
MVN DST,S1	NOT register
MRS DST,PSR	Read PSR
MSR PSR,S1	Write PSR
SWP DST,S1,ADDR	Swap reg/mem word
SWPB DST,S1,ADDR	Swap reg/mem byte
SWI IMM	Software interrupt

S1 = source register
S2IMM = source register or immediate
S3 = source register (when 3 are used)
DST = destination register
D1 = destination register (1 of 2)
D2 = destination register (2 of 2)

ADDR = memory address
IMM = immediate value
REGLIST = list of registers
PSR = processor status register
cc = branch condition

Addressing Modes

- Many instructions have operands (inputs), so where do they come from?
  - Addressing mode specifies this
- Addressing modes
  - Immediate addressing
  - Direct addressing
  - Register addressing
  - Register indirect addressing
  - Indexed addressing
  - Based-index addressing
  - Stack addressing
  - …
- Only some of these are available in typical modern ISAs
  - ARM has many addressing modes
  - Don’t have to use them all, but good to be aware of them

Immediate/Literal Addressing

1. Immediate: ADD r2, r0, #5

cond	f	opcode	rn	rd	Immediate

- Operand comes from the instruction
- Example: 32-bit instruction to move 4 into R1
  - Result is R1 := 4

MOV R1 #4

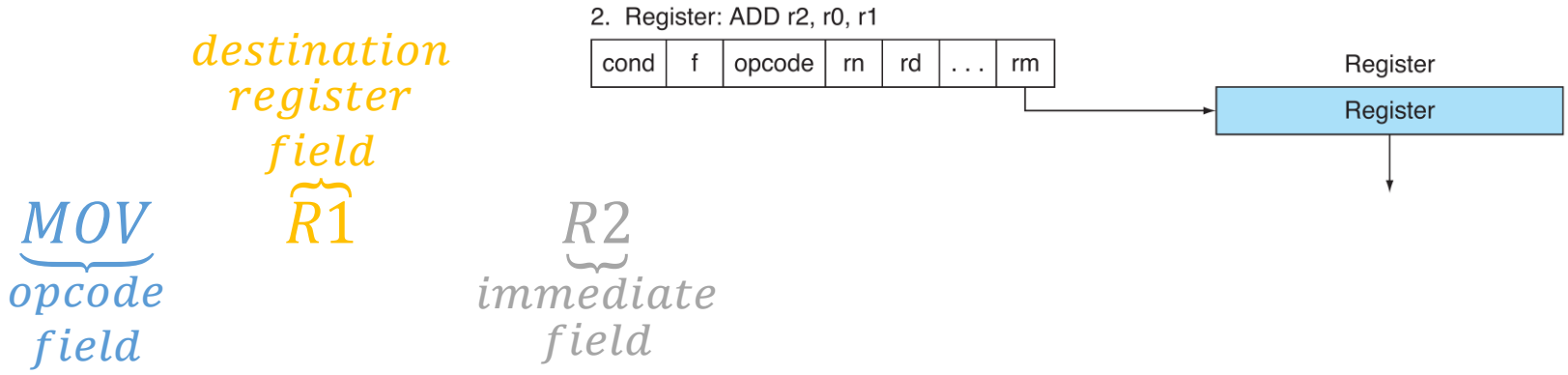
destination register field

MOV (*opcode field*) *R1* #4 (*immediate field*)

- Useful for specifying small integer constants (avoids extra memory access)
- Can only specify small constants (limited by size of immediate field)
  - ARM: typically 8-12-bits

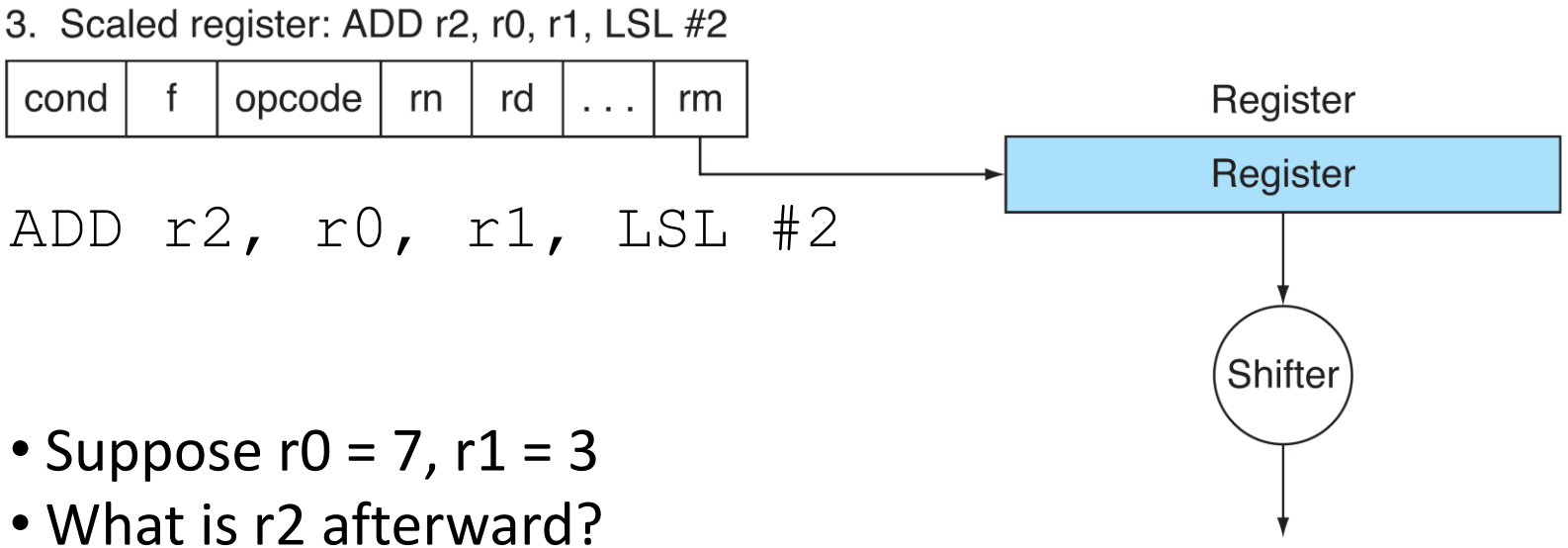
Register/Register-Direct Addressing

- Operand(s) come(s) from register(s)
  - Seen this many times already: ADD R0 R1 R2 does R0 := R1 + R2
  - Also: MOV R1 R2: the destination operand is specified by its register address (Result is R1 := R2)



Scaled Register Addressing

- Uses a value from a register and modifies it (here, LSL is logical shift left, i.e., move number 2 bits left)
  - Equivalent to multiplying by 4 (each bit shift is *2)
- Allows combining a couple operations (efficiency)



- Suppose r0 = 7, r1 = 3
- What is r2 afterward?
  - R2 = 7 + (3*2*2) = 19

Direct/Absolute Addressing

- Operand comes from accessing its full address in memory
- Example: 32-bit instruction to move data from memory location (address) 4 into R1 (result is R1 := MEM[4])

```
MOV  R1  #4
```

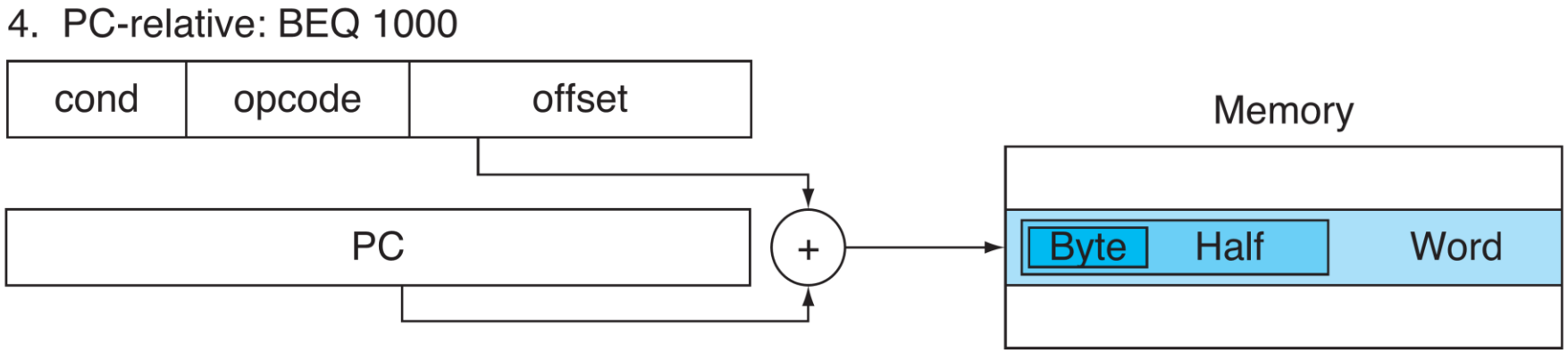
- Useful for specifying global variables
- Problem with example? How many memory locations? How big are the immediate fields?
- While the value at the address can change, the address (location) cannot
- ***Not available in ARM***

Register Indirect Addressing

- Operand comes from memory, with address specified by the value in a register (i.e., by a pointer) or by an immediate
- Example: ARM instruction to copy value from MEM[R4] into R1 (e.g., R1 := MEM[R4])

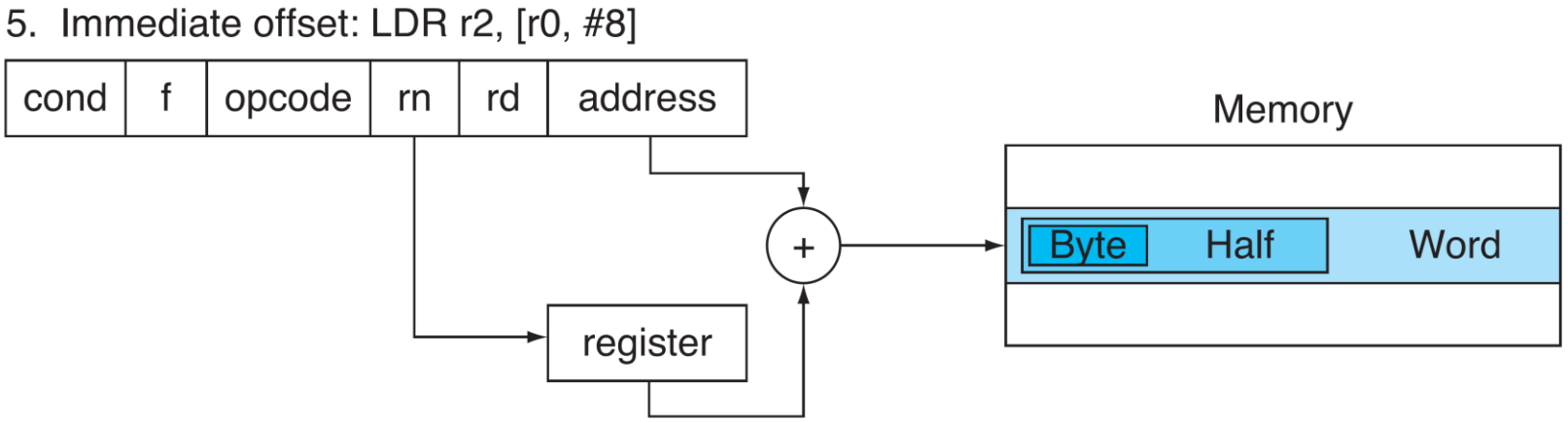
```
LDR R1 [R4]
```

PC-Relative Indirect



- Uses the current PC value with an immediate offset to determine the value
- Example: branch if equal to location PC + 1000
  - Updates PC = MEM[PC + 1000]
- Example: LDR r6, [PC]
  - Updates r6 = MEM[PC]

Indirect with Immediate Offset

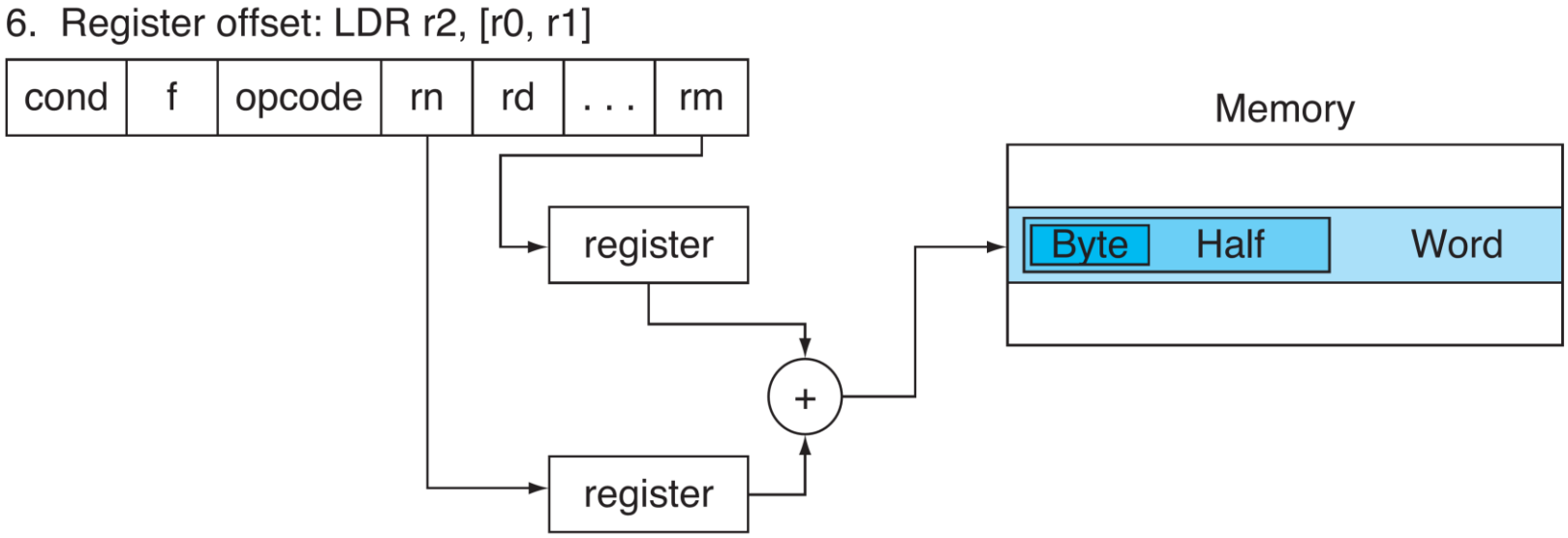


- Uses a register value and an immediate offset
- Example: LDR r2, [r0, #8]
  - Updates r2 = MEM[r0 + #8]

PC-Relative Addressing

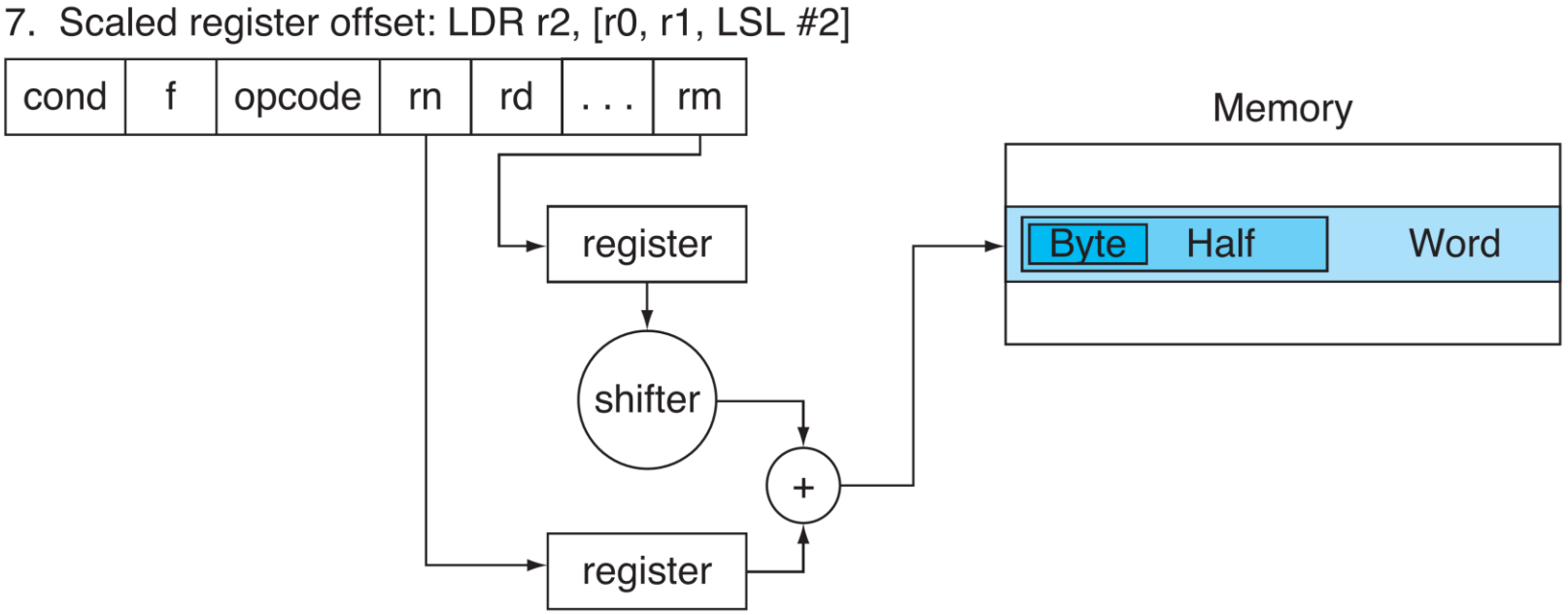
- Same as indexed mode, but register used is the PC: operand comes from MEM[PC + offset]
- Shows up in control flow (branch) instructions
- Note that the offset may be signed (negative)
- Why is this useful?

Indirect Register Offset



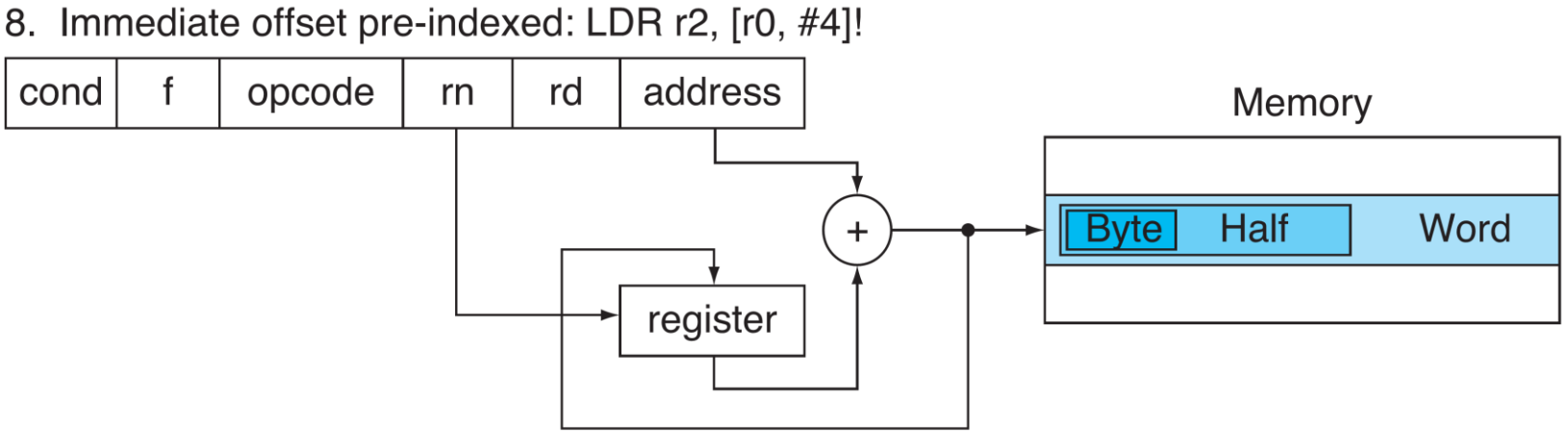
- Uses a register value and another register value as an offset
- Example: LDR r2, [r0, r1]
  - Updates r2 = MEM[r0 + r1]

Indirect Scaled Register Offset



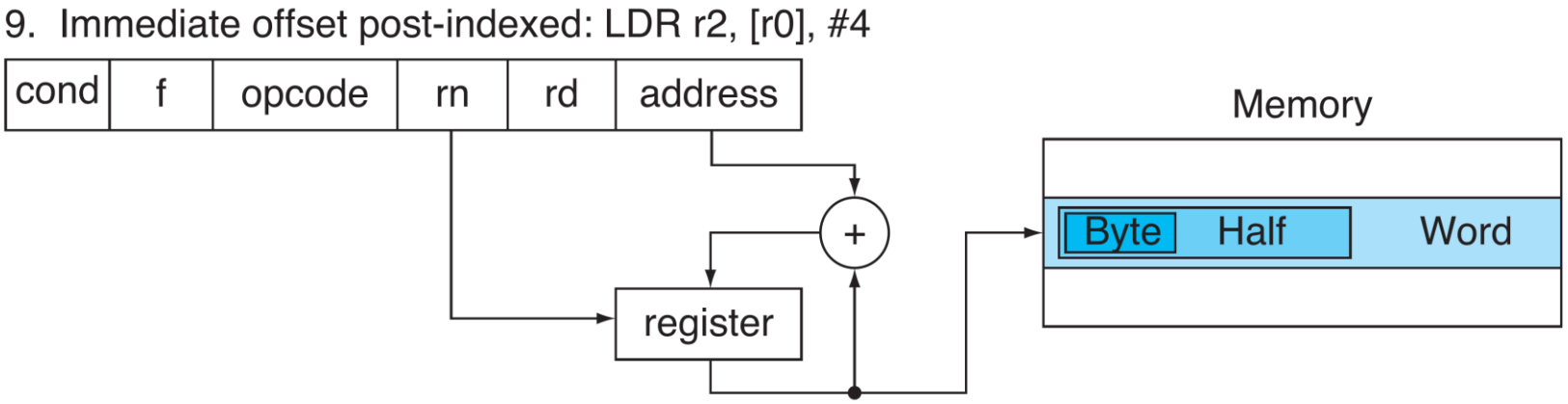
- Uses a register value and another register value as an offset, that is scaled
- Example: LDR r2, [r0, r1, LSL #2]
  - Updates r2 = MEM[r0 + r1*4]
  - As before, LSL #2 multiplies by 4 (two bit shifts left)

Immediate Offset Pre-Indexed



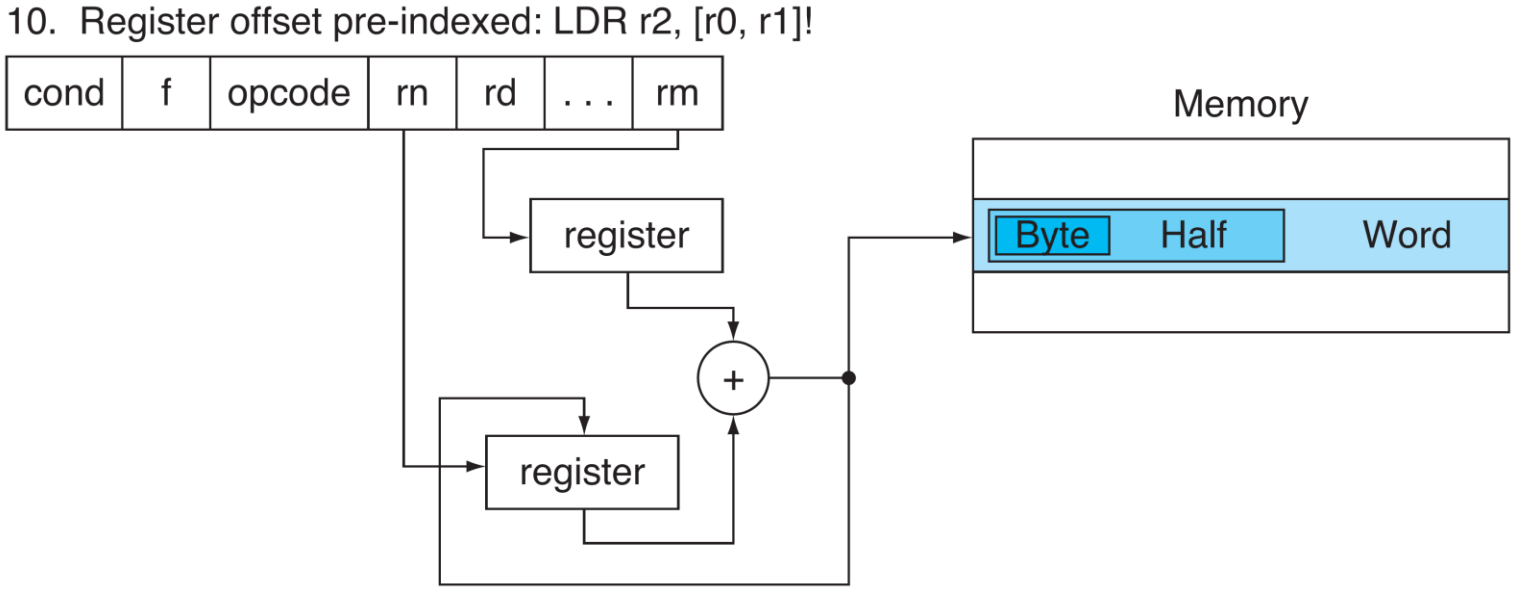
- Uses a register value and an immediate to compute address, and also updates the register offset (before transfer)
- Example: `LDR r2, [r0, #4]!`
  - Updates:
    - `r0 = r0 + 4`
    - `r2 = MEM[r0]` (using the new value)

Immediate Offset Post-Indexed



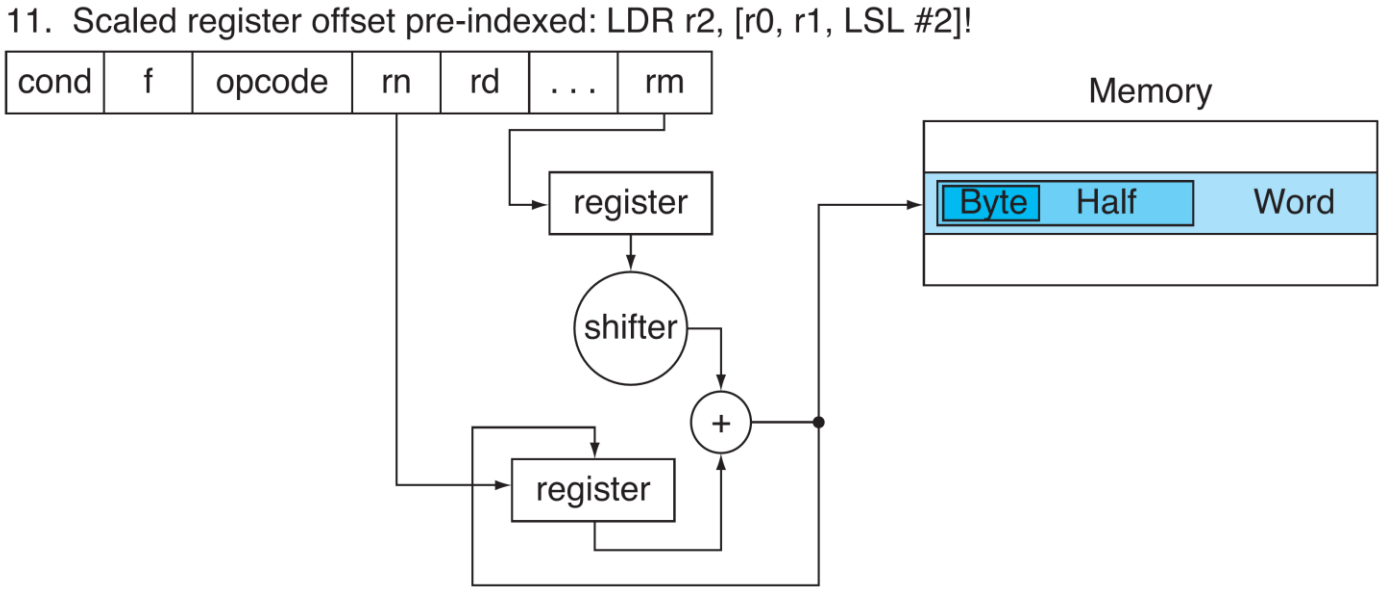
- Uses a register value and an immediate to compute address, and also updates the register offset (after transfer)
- Example: `LDR r2, [r0], #4`
  - Updates:
    - `r2 = MEM[r0]`
    - `r0 = r0 + 4`

Register Offset Pre-Indexed



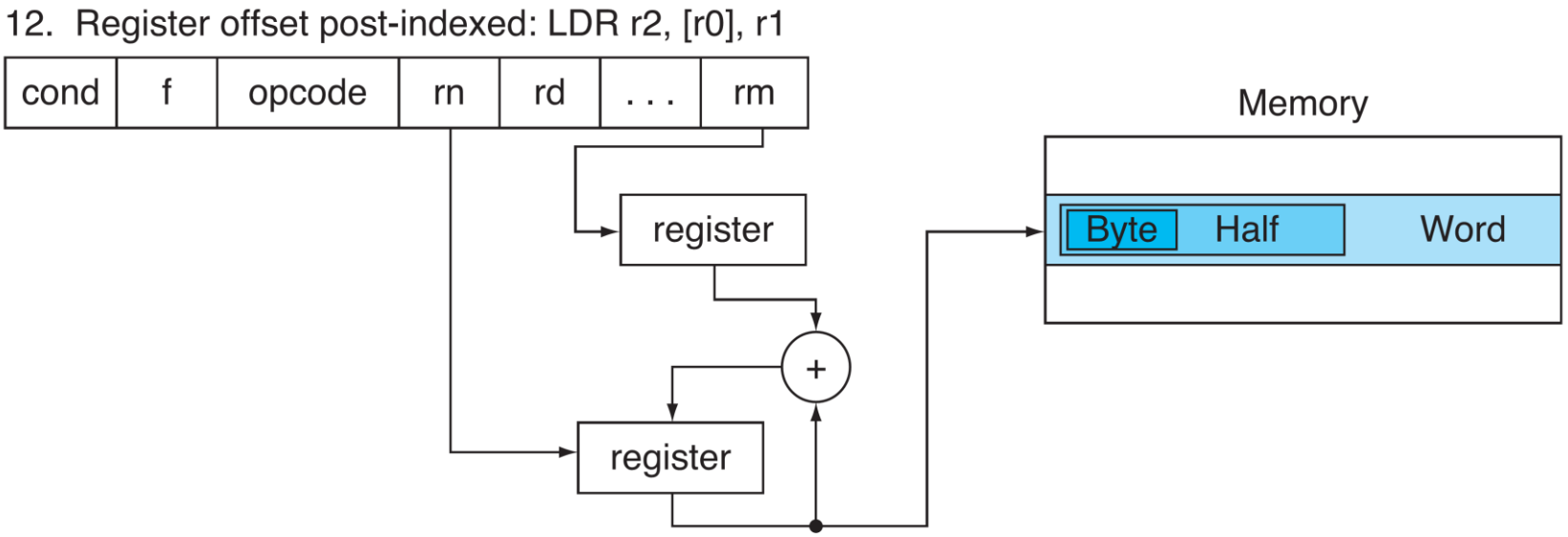
- Uses two registers to compute address, and also updates the register offset (before transfer)
- Example: `LDR r2, [r0, r1]!`
  - Updates:
    - `r0 = r0 + r1`
    - `r2 = MEM[r0]` (using the new value)

Scaled Register Offset Pre-Indexed



- Uses two registers to compute address, and also updates the register offset (before transfer), and scales
- Example: `LDR r2, [r0, r1, LSL #2]!`
  - Updates:
    - `r0 = r0 + r1*4`
    - `r2 = MEM[r0]` (using the new value)

Register Offset Pre-Indexed



- Uses two register values, and also updates the register offset (after transfer)
- Example: `LDR r2, [r0], r1`
  - Updates:
    - `r2 = MEM[r0]`
    - `r0 = r0 + r1`

Stack Addressing

- Operand specified by **stack pointer register, SP**
- Example: reverse Polish notation computation
- Operand comes from MEM[SP]
- SP then usually incremented or decremented depending on order of memory (to accomplish the stack POP)
- ***Not available in ARM***
  - Not typically available in hardware anymore (no registers)
  - Some examples: Java byte-code and JVM

Addressing Modes

- Specify where to get operands (inputs) for instructions
- Why have all these different modes?
- What are some tradeoffs?
- What could machine language instruction formats look like for each of the addressing modes?

Assembly Language

- Assembly: source language is symbolic representation for numerical machine language
  - Assembler (not compiler) used in this case to perform translation
- Pure assembly: one-to-one correspondence between assembly language and machine language
- Practical assembly: extra commands that the assembler takes care of
  - Examples: computing addresses from labels, filling consecutive addreses with zeros, placing strings in memory, etc.

Assembly Language Format

Label	Opcode	Operands	Comments
iloop:	add	r1,r1,#1	@ r1 := r1 + 1
	b	iloop	@ pc := iloop
val:	.byte	0x9F	@ put 0x96 at address val
s:	.asciz	"hello!"	@ put "hello!" at sequential addresses starting at address s

Assembly Instructions vs. Directives

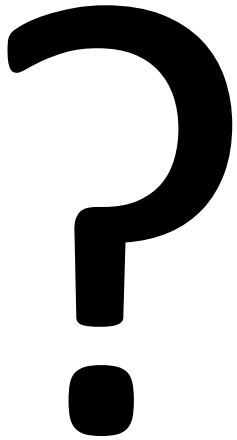
- Instructions
  - Machine language equivalents
- Directives / Pseudoinstructions
  - Special commands given to the assembler that are converted to equivalent machine language during assembly process
  - Used for placing data in memory, etc.

Summary

- Many addressing modes
- Be familiar with types available for architecture you are working with, will influence efficiency (e.g., number of memory accesses, code size, etc.)

Addressing mode	Core i7	OMAP4430 ARM	ATmega168 AVR
Immediate	×	×	×
Direct	×		×
Register	×	×	×
Register indirect	×	×	×
Indexed	×	×	
Based-indexed		×	

Questions?



Translation Example

- Suppose our ISA does not have a multiply instruction
- How can we perform multiplication?
  - Create equivalent sequence of computations yielding the same result
  - Use addition, branch, comparisons, etc.
  - $A * B = \sum_{i=1}^{B} A = \underbrace{A + A + \cdots + A}_{B\ times}$
  - Example: $5 * 9 = \sum_{i=1}^{9} 5 = \underbrace{5 + 5 + \cdots + 5}_{9\ times} = 45$
- Generalizing: this is the basis of all our modern computations
- CPU does not have “visit website, buy shoes“ instruction