Instruction Set Wish List

Fleshing out the parameters of the ISA

Memory Model

Address Range

- 32-bit => 2^{32} or 4G(byte/word) maximum
- Address by word or byte?
- Don’t need full address range (most of our applications will be small)
  - Shortsighted initial views capped addresses at fewer bits
- Keep in mind that address is used for both data and branches

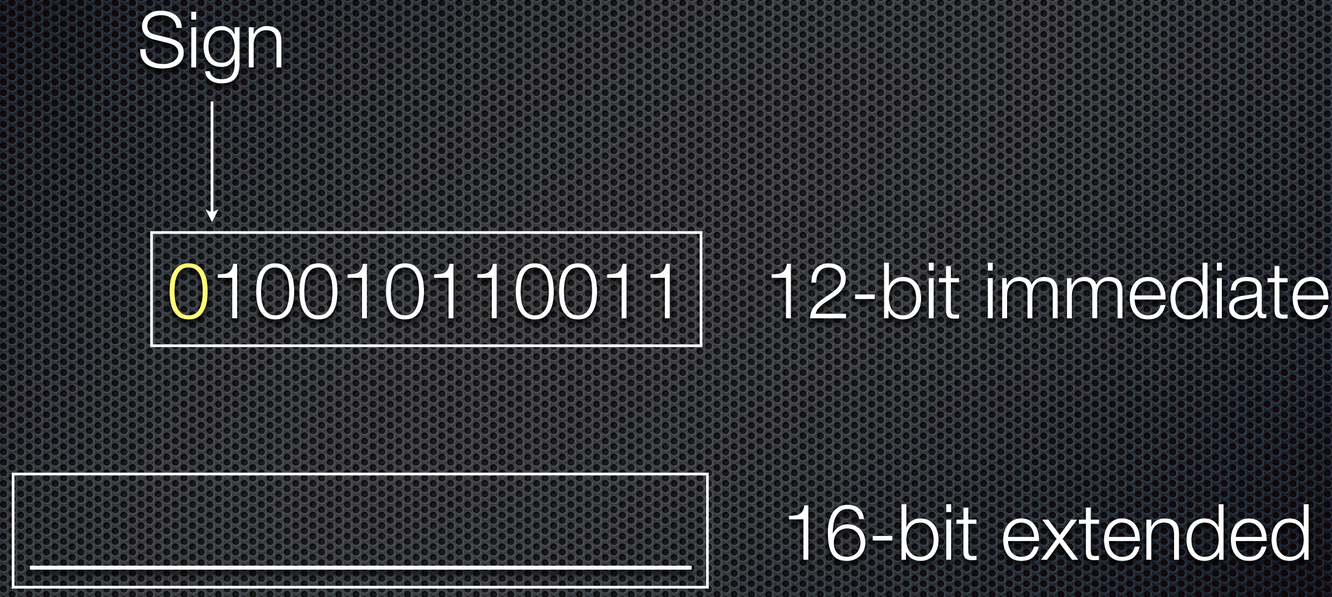
Addressing

- Immediate (operand value in instruction, usually sign-extended)
- PC relative (PC + sign extended immediate)
- Register direct (register number in instruction, operand value in register)
- Register indirect (register number in instruction, address in register)
- Base + index
  - One register has base address, second has index, added to get address
- Base + index + offset
  - Adds immediate offset, sign extended, to base and index values
- Many others (e.g., pre/post inc/dec, memory indirect, etc.)

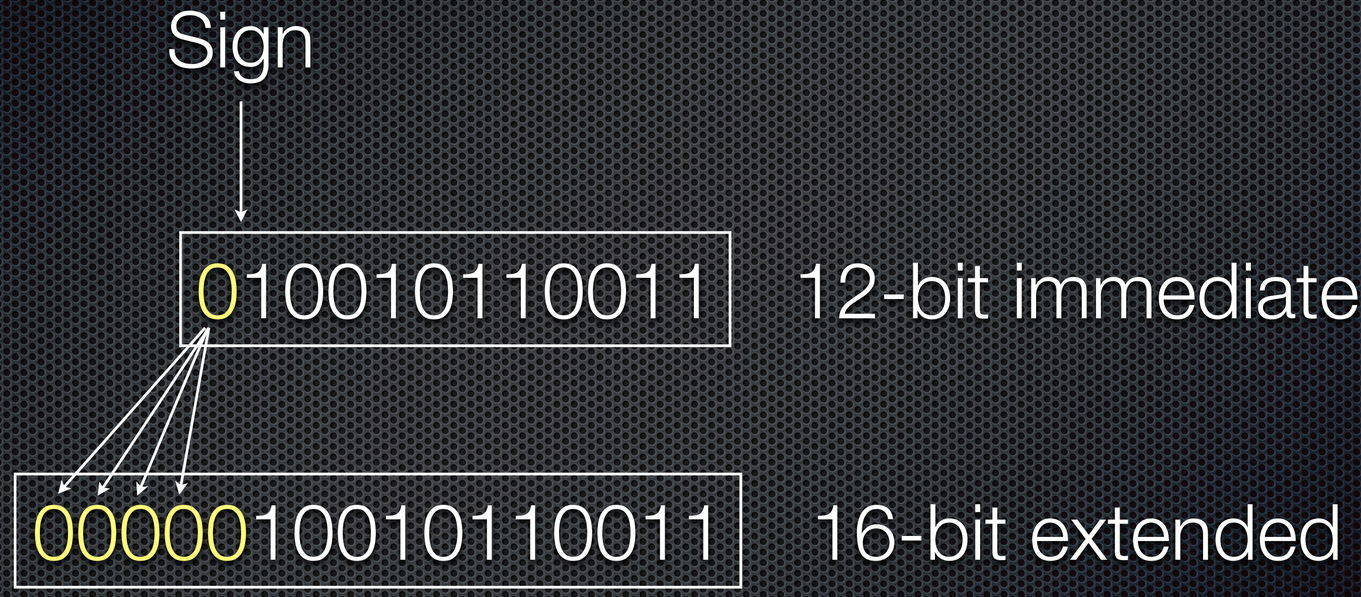
Addressing Modes

- Immediate: Value in an instruction field
  - Often with sign extend or shift

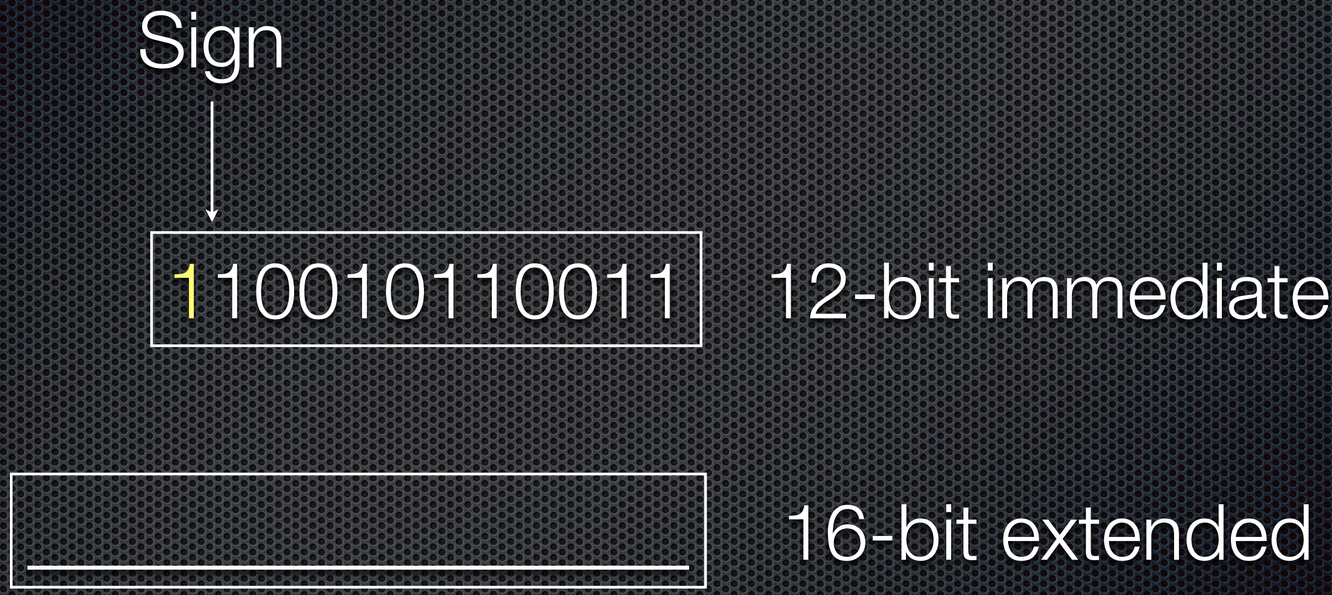
Sign Extension



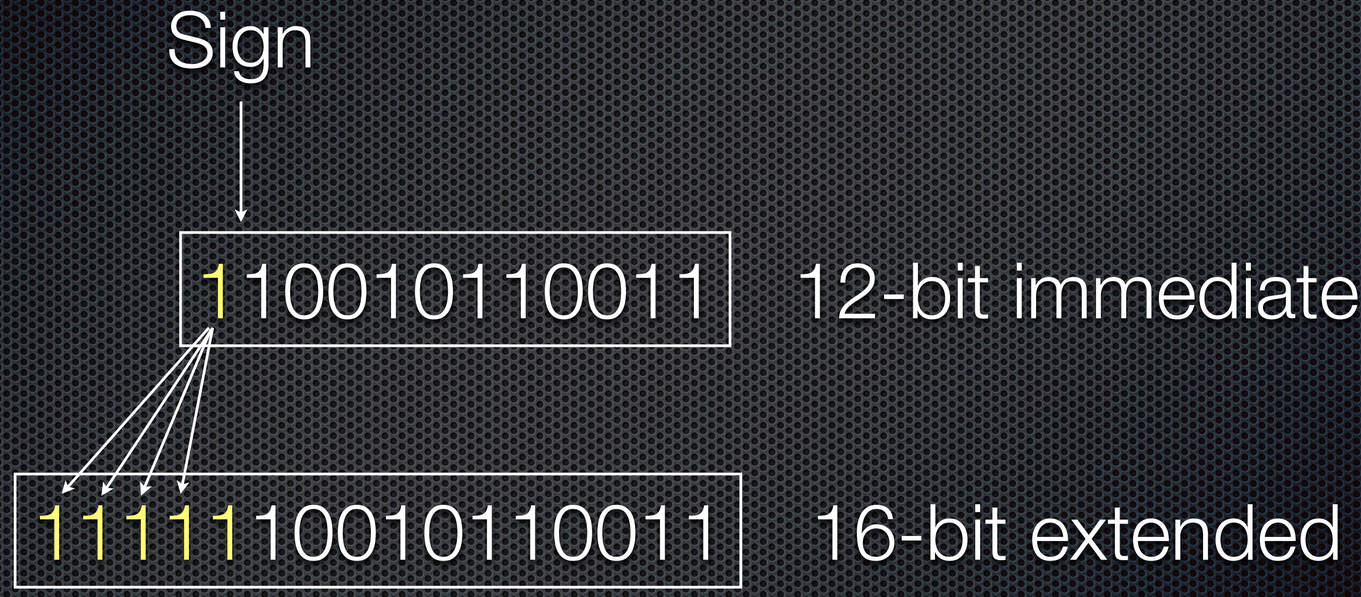
Sign Extension



Sign Extension



Sign Extension

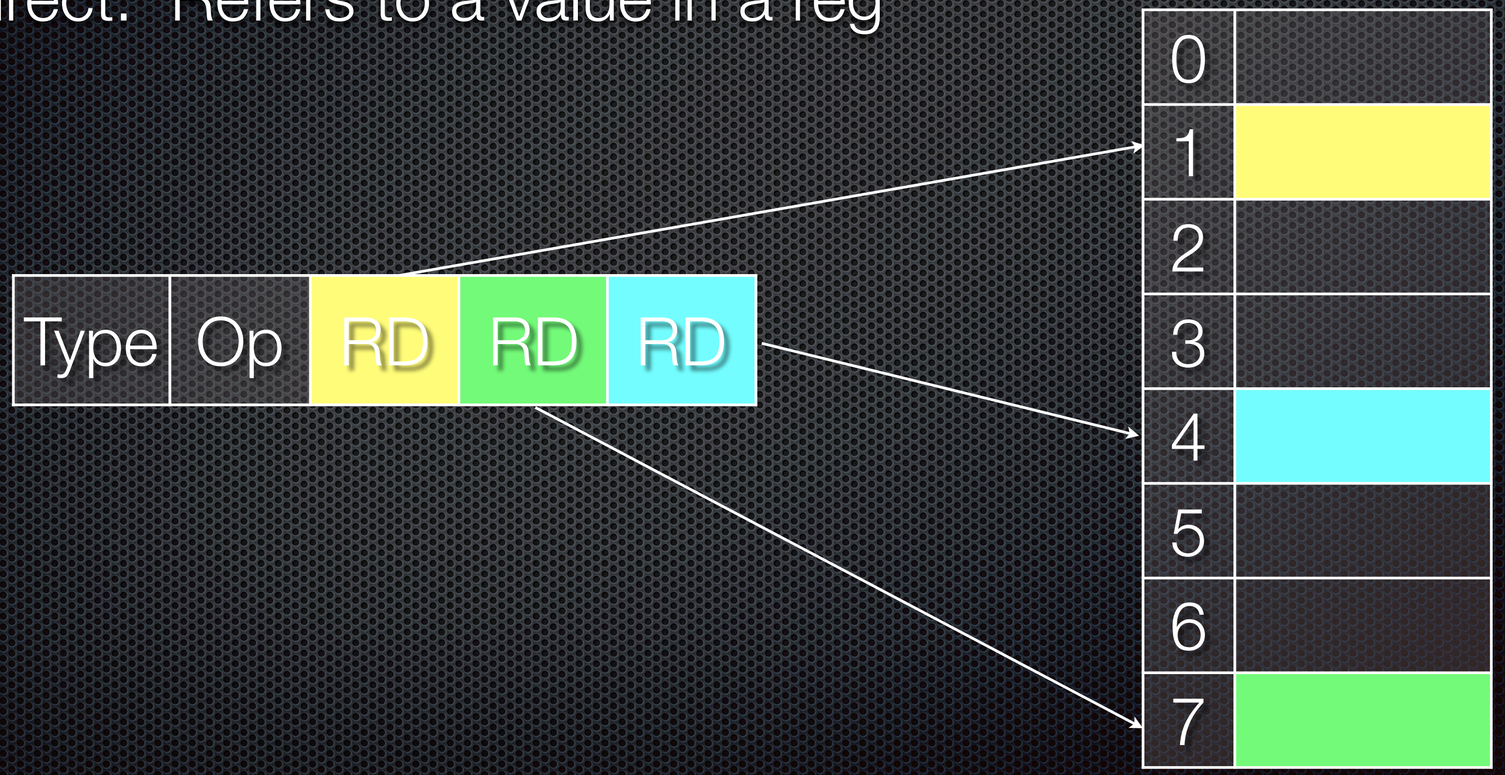


Addressing Modes

- PC+ Immediate offset: add sign extended immediate value to PC to get address

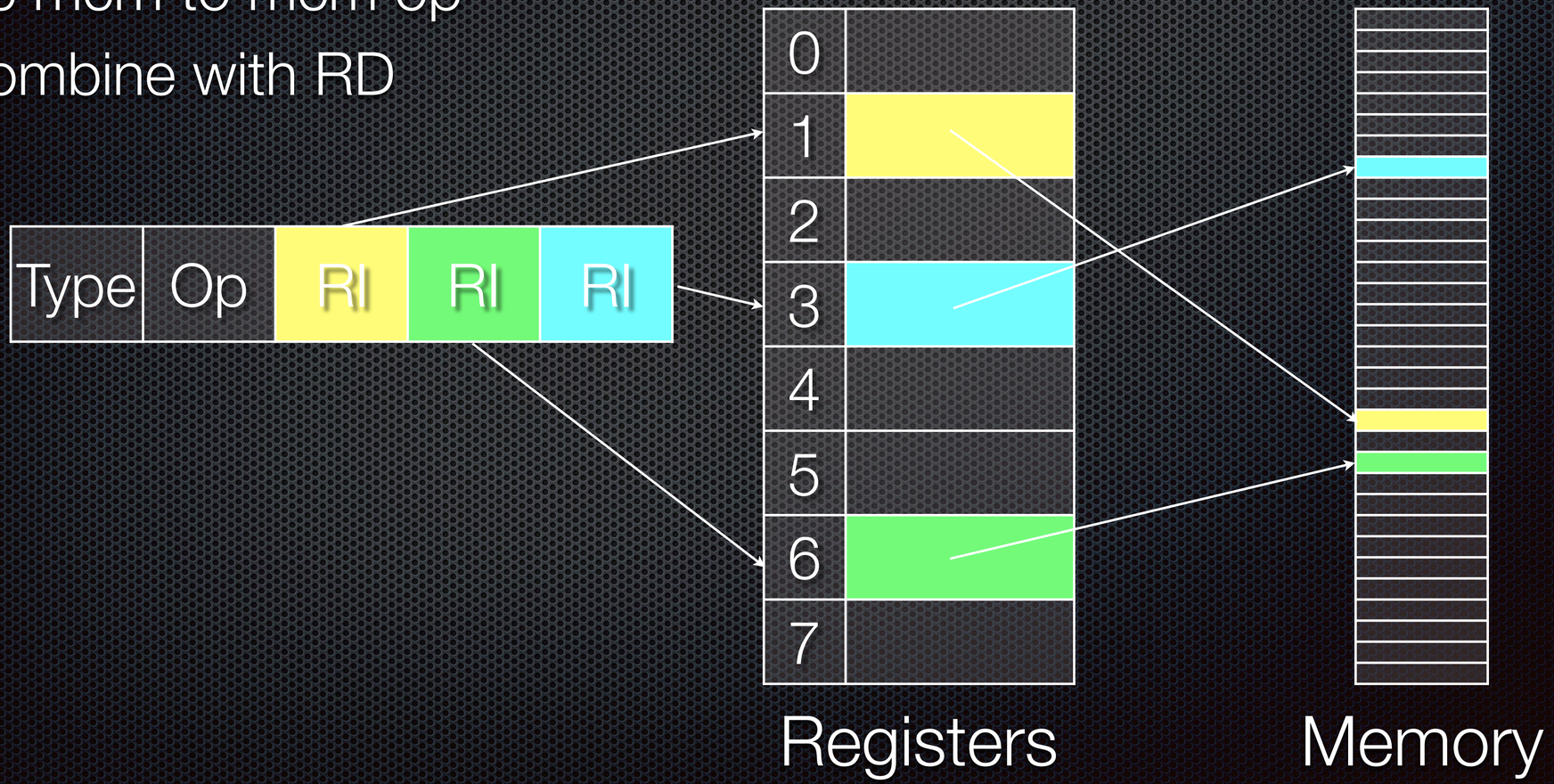
Addressing Modes

- Register direct: Refers to a value in a reg



Addressing Modes

- Register indirect: Reg contains memory address
- Can be mem to mem op
- May combine with RD

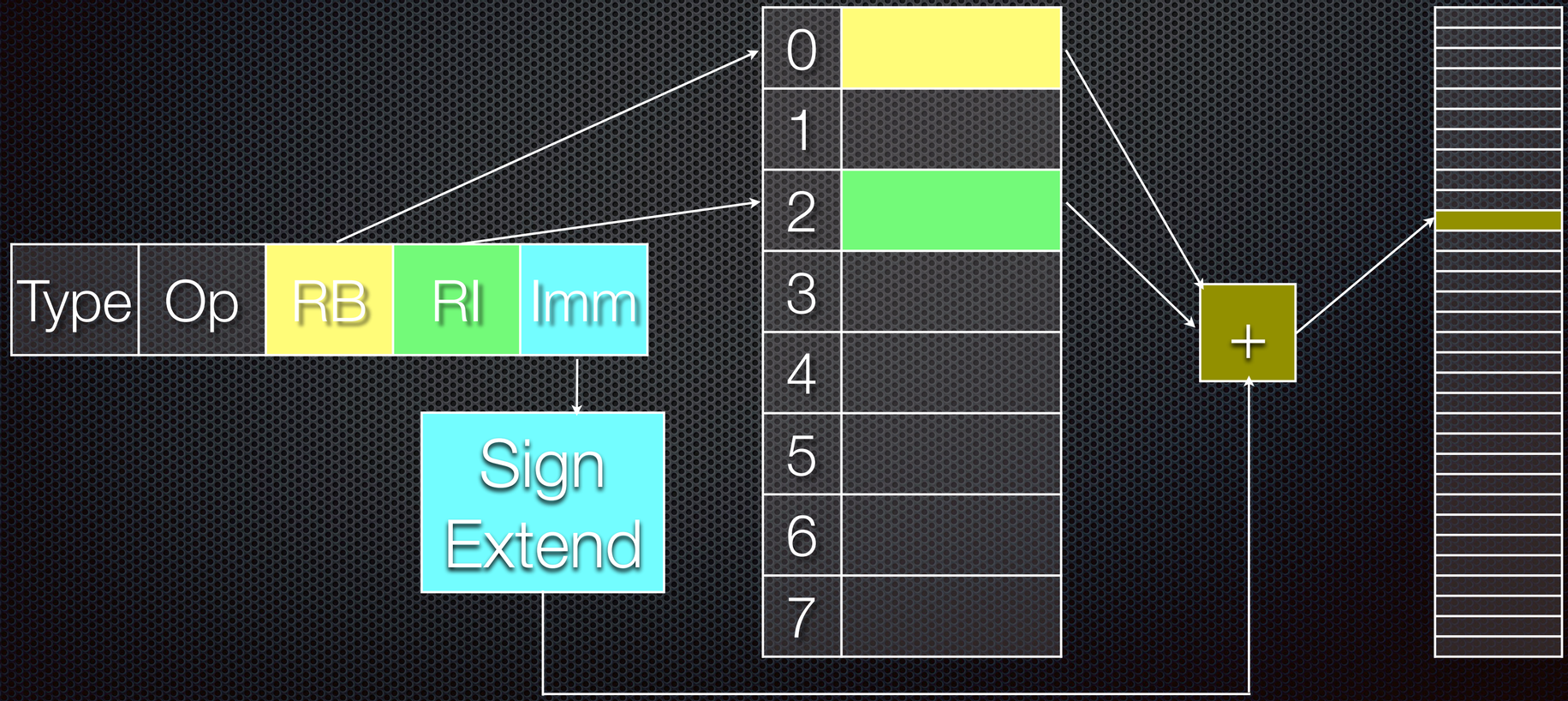


Addressing Modes

- Register indirect base + index
- Usually load/store
- R_{Source}/R_{Dest} not shown

Addressing Modes

Register indirect base + index + immediate offset: RB + RI + sign extended immediate is memory address



More Modes

- Widths (load byte, half word, etc.)
- Variations of shifting/offset (moving smaller values to/from byte positions)
- Pre/Post Increment/Decrement: automatic operation on address register
- Memory indirect: Value in mem is address

More Modes

- Widths (load byte, half word, etc.)
- Variations of shifting/offset (moving smaller values to/from byte positions)
- Pre/Post Increment/Decrement: automatic operation on address register
- Memory indirect: Value in mem is address

Use of Modes

- Register direct: ALU, Move, Compare
- Register indirect: Jumps, Subroutine calls/returns
- PC + Immediate Offset (PC Relative): Branches
- Base ((+Index) + Offset): Load and store
- Others are useful but nonessential
- Associating each mode with an instruction type simplifies instruction formats

Instructions: Load/Store

- Load and store word with register
- Accessing smaller units requires addressing those units, and alignment of values
- Alternative is mask (AND) and shift in ALU
- Need separate instructions for different banks (e.g., LoadInt, LoadFP, LoadVector)

Control

- Conditional branches
  - Test/branch approaches
- Jumps and unconditional branches
- Long/short jumps/branches
- Predicated instructions
- Subroutine jumps
- Interrupts, supervisor state
- I/O model (programmed, memory map)

Instructions: Branch/Jump

- Conditional
  - Test and branch (branch does test)
  - Branch on condition (condition is set by another instruction, either automatically as a side effect, or explicitly)
  - Branch on flag (prior instruction sets flag)
  - Branch: PC relative
- Unconditional:
  - Jump: reg indirect, Subroutine call (jump with save PC), Return (Jump via return reg)
- Predication: Instruction may turn into no-op

Condition Codes

- Integer: Carry out/overflow, zero, nonzero, sign (+/-), special state
- Floating Point: Overflow, underflow, NaN, denormalized, infinity, zero, nonzero, mantissa sign, exponent sign
- Vector: Equality, exceptions

Instructions: Subroutine

- Need way to return
- Simplest is store PC++ in return register
- Return is Jump indirect using return register
- Return register may be special or general
- Other approaches try to automate stack (e.g., register windowing)

Instructions: Integer ALU

- Add, Sub, *Mul, Div, Mod, Reverse Sub*
- AND, OR, NOT, XOR
- Compare, Set Flag
- Shifts
- Move register to register (also Swap)

Instructions: Shifts

- Multiple Forms
  - Logical: bits fall off ends, 0 shifts in
  - Rotate: bits recycle to opposite end
  - Arithmetic:
    - Right: sign bit is extended, right bit falls off
    - Left: 0 shifts in at left, bit N-1 falls off, bit N (sign) remains unchanged

Floating Point

- Load/Store
- Add, Sub, Mul, Div, others
- Compare/Status
- Many exceptions: Overflow, underflow, divide by 0

Now the Fun Part

- Given the wish list of features, pack them into the instruction format(s)
- Need to be organized

Group by Type

- Example: ALU, Memory, Branch, Jump
- Example: Integer, FP, Logical, Memory, I/O Branch, Jump, Predicated
- Each type usually has a unique format
- Simplifies layout and decoding

Format: Type Field

- Having a field in common across all types that specifies the type/format, simplifies decoding
- Typically 3 or 4 bits
- One value may have subtypes in another field

Fields by Type

- For each type, determine the fields it needs
  - Example: ALU has three operand fields
  - Branch has condition and PC relative fields
  - Memory has load/store, mode, and address regs
- What type of info in each field (register number, immediate value, offset, address)

Format: Opcode

- Need not be consistent across types
  - Many subtypes of Int, logical; only a few FP
- Small number of bits -- logically an extension of the type field
- In simulator, becomes input to jump table

Bin Packing

- Squeeze fields into instruction space
- May need to change number of registers or number of operands
- Maybe reduce set of operations
- Tempting to fill holes, but don’t

Assigning Codes

- Determine binary values for
  - Instruction types
  - Operation codes
  - Comparisons
  - I/O devices or memory-mapped locations
- Document formats, values, and semantics

Example Instruction (ARMv7 manual)

A7.7.4 ADD (register)

ADD (register) adds a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T3 ARMv7-M

```
ADD{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	0	0	0	S	Rn				(0)	imm3			Rd				imm2		type		Rm			

Further Documentation

Assembler syntax

`ADD{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}`

where:

`S`	If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.
`<c><q>`	See *Standard assembler syntax fields* on page A7-207.
`<Rd>`	Specifies the destination register. If `<Rd>` is omitted, this register is the same as `<Rn>` and encoding T2 is preferred to encoding T1 if both are available. This can only happen inside an IT block. If `<Rd>` is specified, encoding T1 is preferred to encoding T2. If `<Rm>` is not the PC, the PC can be used in encoding T2.
`<Rn>`	Specifies the register that contains the first operand. If the SP is specified for `<Rn>`, see *ADD (SP plus register)* on page A7-227. If `<Rm>` is not the PC, the PC can be used in encoding T2.
`<Rm>`	Specifies the register that is optionally shifted and used as the second operand. The PC can be used in encoding T2.
`<shift>`	Specifies the shift to apply to the value read from `<Rm>`. If `<shift>` is omitted, no shift is applied and all encodings are permitted. If `<shift>` is specified, only encoding T3 is permitted. The possible shifts and how they are encoded are described in *Shifts applied to a register* on page A7-212.

Instruction Set Specification

Defining the programming contract

Where we are

- Instructions defined abstractly
- Grouped into types
- Types are assigned values
- Next step is designing instruction formats
- Assign codes for operations

Field type: Instruction type

- Usually fixed in position for all types
  - Makes decoding easier
- Typically 2 to 4 bits (4 to 16 types)
- Can have a type that indicates additional bits specify a sub-type
  - Example: 3-bit type, value 7 indicates next 2 bits extend the type

Field type: Opcode

- Often starts in same position for all types, but may differ in length
  - Example: Many more ALU ops than branch ops
- Typically 3 to 6 bits
  - May have values that indicate additional bits extend the opcode (typically used when type field is small)

Field type: Register addr

- $\log_2(R)$ bits, where R= # registers
- One field per operand
- Not every instruction type needs 3 fields
  - Two or one address architectures use fewer
  - Some operations may have 4 fields
- Need not be adjacent

Field type: Immediate

- Made up of “left-over” bits
- Size may vary with format
- Usually sign extended by operations

Other Fields

- Condition (may be in branch only, or predicate for all)
- Set/Don’t set condition (a la Arm)
- Shift amount (how many positions)
- “Micro-op” formatting

Endianness

- Byte order within a word
- Given value 0x12345678, what order should the bytes (12, 34, 56, 78) appear in a word?
- 12 is high order, 78 is low order

Endianness

If you say:

```
Byte address:   0    1    2    3
Value:                12  34  56  78
```

Then you are in favor of Little-endian addressing
(low order byte comes at the end)

Endianness

If you say:

Byte address: 0 1 2 3
Value: 78 56 34 12

Then you are in favor of Big-endian addressing
(high order byte comes at the end)

What About Strings?

Given ASCII string (8-bit subset of Unicode): “ABCD”

Normal placement is first character in low-order byte, last character in high-order byte

What About Strings?

Given ASCII string (8-bit subset of Unicode): “ABCD”
Little endian byte address 3 holds the low-order byte,
so all of you “Little-endians” end up with:

Byte address:	0	1	2	3
Value:	D	C	B	A

First character, ‘A’, is in low order byte,
last character, ‘D’, is in high order byte

What About Strings?

Given ASCII string (8-bit subset of Unicode): “ABCD”
Big endian byte address 0 holds the low-order byte,
so “Big-endians” end up with:

Byte address:	0	1	2	3
Value:	A	B	C	D

First character, ‘A’, is in low order byte,
last character, ‘D’, is in high order byte

Does Endianness Matter?

- Within an architecture, it’s just a matter of wiring, and everything is consistent
- Transferring files between systems of different endianness complicates data sharing

Modes

- Endianness can be modal
- Can have alternate instruction set modes
- Typical modes are security levels
  - User/Supervisor, additional levels for VM, etc.

Interrupts

- Asynchronously cause jump to handler
- Usually a low area of memory contains a table of jumps
- Interrupt type N jumps to Nth element of jump table, causing jump to handler (saves PC in separate return register)
- Need to be able to turn off
- Often supported by supervisor mode

Virtual Memory

- Generally requires supervisor mode
  - Privileged instructions to manage memory map
- Defines virtual to physical address mapping
- Can be paged, segmented, combination, or a hierarchy
- Not required for simulator project

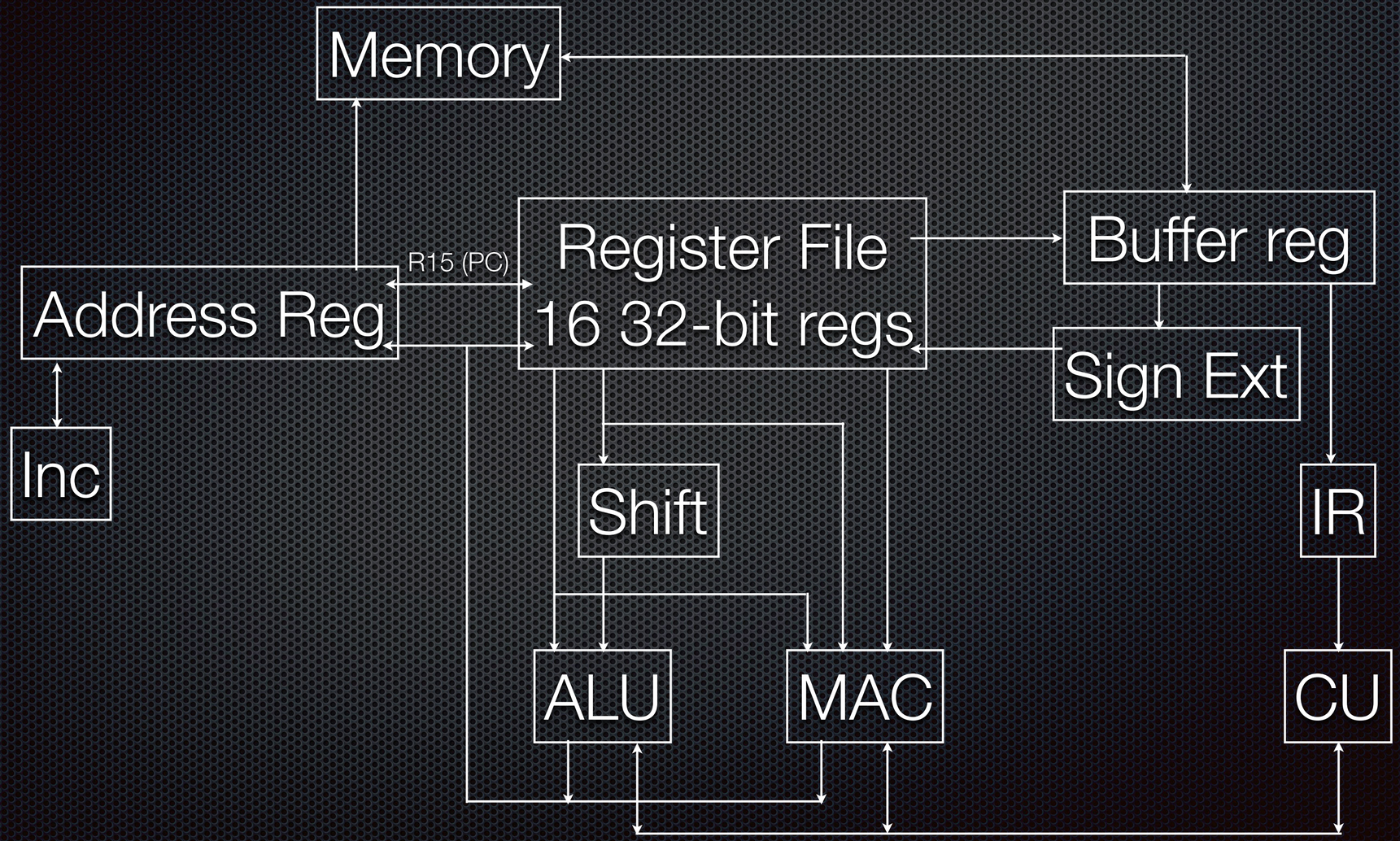
Microarchitecture Ops

- Sometimes need to interact with aspects of the microarchitecture
- Typically need ability to force cache lines to flush to memory for I/O
- May have branch hints for predictor warm-up
- May have privileged instructions for managing other cached state (TLB, branch predictor)

Arm Architecture

Example Embedded RISC Processor

ARM Organization



Arm Registers

- 32-bits each
- R0 - R12 General Purpose
- R13 Stack Pointer
- R14 Link Register (subroutine return)
- R15 Program Counter
- CPSR Status Register
  - Condition codes, overflow, etc.

ARM Data Types

- 8, 16, 32, 64-bit integers (depending on model)
- Floating point and vector supported by some versions
- Can be big endian or little endian, as set by user

ARM Instruction Formats

- Bits 31-28: Condition Code
- Bits 27-25: Instruction type
- Types 000 - 001:
  - Bits 24-21: Opcode
  - Bit 20: Condition code update flag
  - Bits 19-16: Source register 1
  - Bits 15-12: Destination register

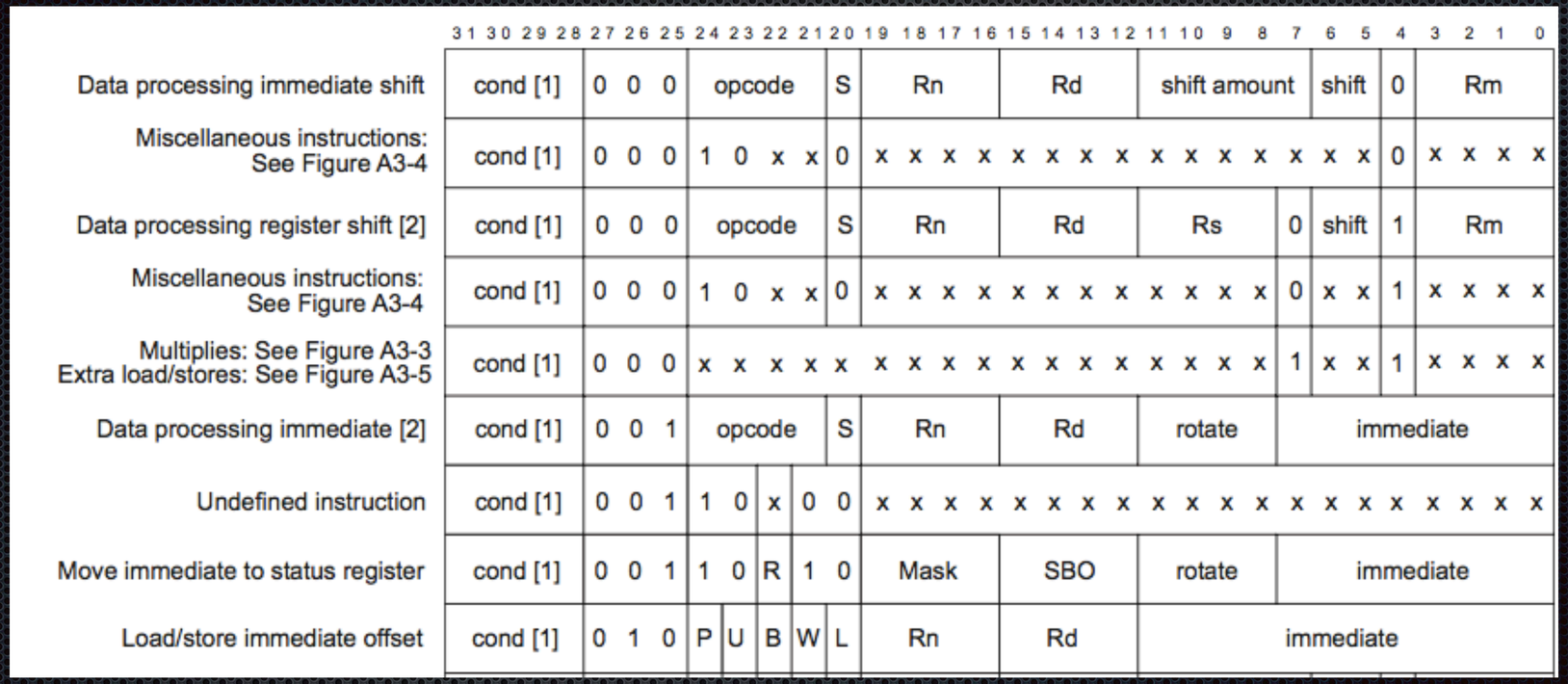
Type 000

- Bits 6-5: Shift type
- Bits 3-0: Source register 2
- Immediate Shift Subset (Bit 4 = 0)
  - Bits 11-7: Shift amount
- Register Shift Subset (Bit 4 = 1)
  - Bits 11-8: Register with shift count
  - Bit 7 = 0

Other Types

- 001: Data processing with immediate operand (7-0) and rotate(11-8)
- 010: Load/store with immediate offset
- 011: Load/store with register offset
- 100: Load/store multiple
- 101: Branch or branch with link

Formats



	31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
Data processing immediate shift	cond [1]	0 0 0	opcode	S	Rn	Rd	shift amount		shift	0	Rm
Miscellaneous instructions: See Figure A3-4	cond [1]	0 0 0	1 0 x x	0	x x x x x x x x x x x x				x x	0	x x x x
Data processing register shift [2]	cond [1]	0 0 0	opcode	S	Rn	Rd	Rs	0	shift	1	Rm
Miscellaneous instructions: See Figure A3-4	cond [1]	0 0 0	1 0 x x	0	x x x x x x x x x x x x			0	x x	1	x x x x
Multiplies: See Figure A3-3 Extra load/stores: See Figure A3-5	cond [1]	0 0 0	x x x x	x	x x x x x x x x x x x x			1	x x	1	x x x x
Data processing immediate [2]	cond [1]	0 0 1	opcode	S	Rn	Rd	rotate	immediate			
Undefined instruction	cond [1]	0 0 1	1 0 x	0 0	x x x x x x x x x x x x			x x x x x x x x x x x x			
Move immediate to status register	cond [1]	0 0 1	1 0 R	1 0	Mask	SBO	rotate	immediate			
Load/store immediate offset	cond [1]	0 1 0	P U B W	L	Rn	Rd	immediate				

More Formats

Load/store register offset	cond [1]	0	1	1	P	U	B	W	L	Rn				Rd				shift amount				shift		0	Rm						
Media instructions [4]: See Figure A3-2	cond [1]	0	1	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	1	x	x	x	x			
Architecturally undefined	cond [1]	0	1	1	1	1	1	1	1	x	x	x	x	x	x	x	x	x	x	x	x	1	1	1	1	x	x	x	x		
Load/store multiple	cond [1]	1	0	0	P	U	S	W	L	Rn				register list																	
Branch and branch with link	cond [1]	1	0	1	L	24-bit offset																									
Coprocessor load/store and double register transfers	cond [3]	1	1	0	P	U	N	W	L	Rn				CRd				cp_num				8-bit offset									
Coprocessor data processing	cond [3]	1	1	1	0	opcode1				CRn				CRd				cp_num				opcode2		0	CRm						
Coprocessor register transfers	cond [3]	1	1	1	0	opcode1			L	CRn				Rd				cp_num				opcode2		1	CRm						
Software interrupt	cond [1]	1	1	1	1	swi number																									
Unconditional instructions: See Figure A3-6	1	1	1	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Condition Codes

Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N set and V set, or N clear and V clear (N == V)
1011	LT	Signed less than	N set and V clear, or N clear and V set (N != V)
1100	GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0,N == V)
1101	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
1110	AL	Always (unconditional)	-
1111	-	See *Condition code 0b1111*	-

DP Instructions

Opcode	Mnemonic	Operation	Action
0000	AND	Logical AND	Rd := Rn AND shifter_operand
0001	EOR	Logical Exclusive OR	Rd := Rn EOR shifter_operand
0010	SUB	Subtract	Rd := Rn - shifter_operand
0011	RSB	Reverse Subtract	Rd := shifter_operand - Rn
0100	ADD	Add	Rd := Rn + shifter_operand
0101	ADC	Add with Carry	Rd := Rn + shifter_operand + Carry Flag
0110	SBC	Subtract with Carry	Rd := Rn - shifter_operand - NOT(Carry Flag)
0111	RSC	Reverse Subtract with Carry	Rd := shifter_operand - Rn - NOT(Carry Flag)
1000	TST	Test	Update flags after Rn AND shifter_operand
1001	TEQ	Test Equivalence	Update flags after Rn EOR shifter_operand
1010	CMP	Compare	Update flags after Rn - shifter_operand
1011	CMN	Compare Negated	Update flags after Rn + shifter_operand
1100	ORR	Logical (inclusive) OR	Rd := Rn OR shifter_operand
1101	MOV	Move	Rd := shifter_operand (no first operand)
1110	BIC	Bit Clear	Rd := Rn AND NOT(shifter_operand)
1111	MVN	Move Not	Rd := NOT shifter_operand (no first operand)

ARM Manual

- Available on 335 course page:
- Arm v7 Reference Manual
- https://people.cs.umass.edu/~weems/homepage/courses/cmpsci-335.html

Team Homework

- For Wednesday 2/13, complete ISA description, including register set description
  - Include format diagrams, encodings
  - Explain all instructions, esp. load/store, branch
  - Use ARM manual as guide for descriptions
- This is the draft of your ISA report

Teamwork

Remember to work as a team!