

Introduction to Assembly Language

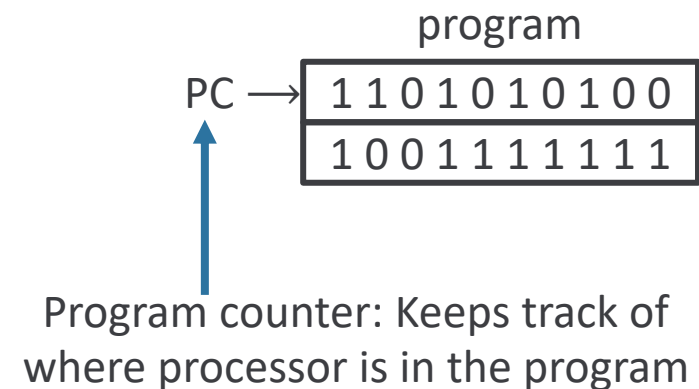
ECE 3140/CS 3420 — EMBEDDED SYSTEMS

What is assembly language?

- **Assembly code:** Human-readable, quasi-isomorphic translation of machine code
 - Ok, but what is machine code?
- **Machine code:** Binary-encoded instructions describing a program
 - Directly executable by the processor

Machine code example (made up)

- Processor:
 - Fetches next instruction from program
 - Decodes instruction
 - Executes according to instruction
 - Rinse and repeat



Machine code example (made up)

- Assume:
 - 16 different instructions
 - 8 registers to store data (also PC, Z)
 - Destination register is also source operand
- Processor:
 - Fetches next instruction from memory
 - Decodes instruction
 - Executes according to instruction
 - Rinse and repeat

PC →

| |
|---------------------|
| 1 1 0 1 0 1 0 1 0 0 |
| 1 0 0 1 1 1 1 1 1 1 |

| op | src1/dst | src2 |
|---------|----------|-------|
| 1 1 0 1 | 0 1 0 | 1 0 0 |

$R[\text{dst}] \leftarrow R[\text{src1}] - R[\text{src2}];$
 $Z \leftarrow R[\text{dst}] == 0; \text{PC} \leftarrow \text{PC} + 1$

| op | offset |
|---------|-------------|
| 1 0 0 1 | 1 1 1 1 1 1 |

$\text{PC} \leftarrow Z ? \text{PC} + 1 : \text{PC} + \text{offset}$

Assembly equivalent (made up)

- Assume:
 - 16 different instructions
 - 8 registers to store data (also PC, Z)
 - Destination register is also source operand
- Processor:
 - Fetches next instruction from memory
 - Decodes instruction
 - Executes according to instruction
 - Rinse and repeat

| | |
|-------|-----------|
| next: | SUB R2,R4 |
| | BNZ next |

| op | src1/dst | src2 |
|-----|----------|------|
| SUB | 2 | 4 |

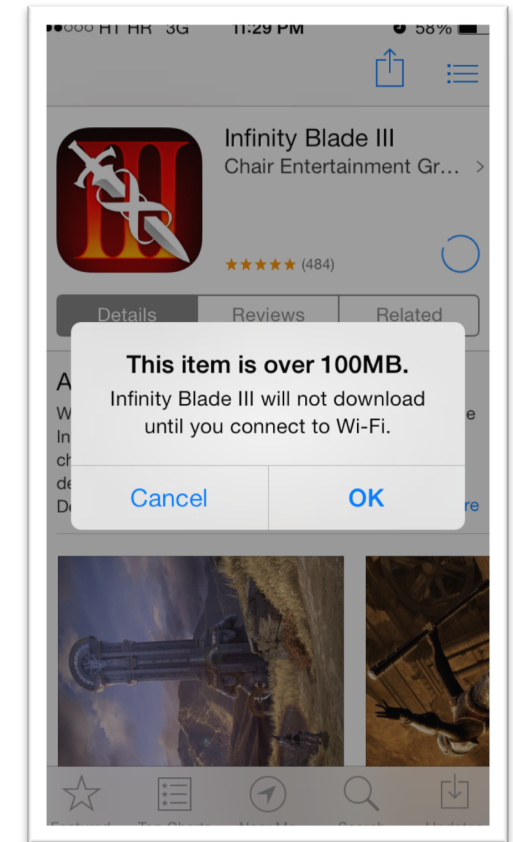
$R2 \leftarrow R2 - R4; Z \leftarrow R2 == 0; PC \leftarrow PC + 1$

| op | offset |
|-----|--------|
| BNZ | -1 |

$PC \leftarrow Z ? PC + 1 : PC - 1$

Who programs in assembly?

- Nowadays, mostly people that enjoy pain and suffering
 - Ok, some low-level tasks best in assembly
- ECE 3140/CS 3420 students (for a few weeks at least)
- Compilers/interpreters *extremely good* at generating fast machine code from high-level languages
 - Tendency for bloated executables (e.g., libraries) →
 - Not always fastest (e.g., critical code block)



iDownloadBlog

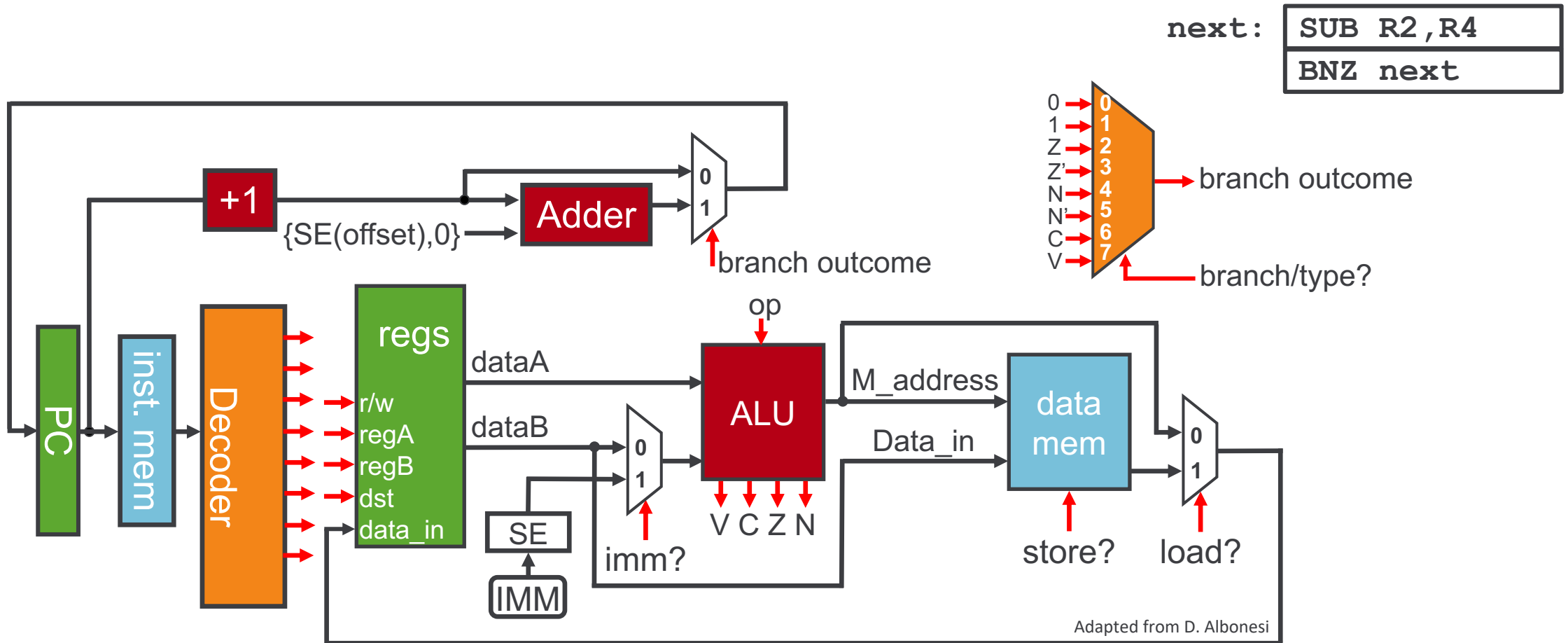
So why study assembly?

- Understand **hardware-software interface**
 - What functionality does the hardware provide?
 - How are high-level language constructs supported?
 - Subroutines, recursion
 - How are system services provided?
 - Dynamic allocation of variables
 - Interaction with I/O devices
 - Multitasking
- Build “bare-metal” (embedded) systems
 - Minimize code bloat; speed up critical code blocks

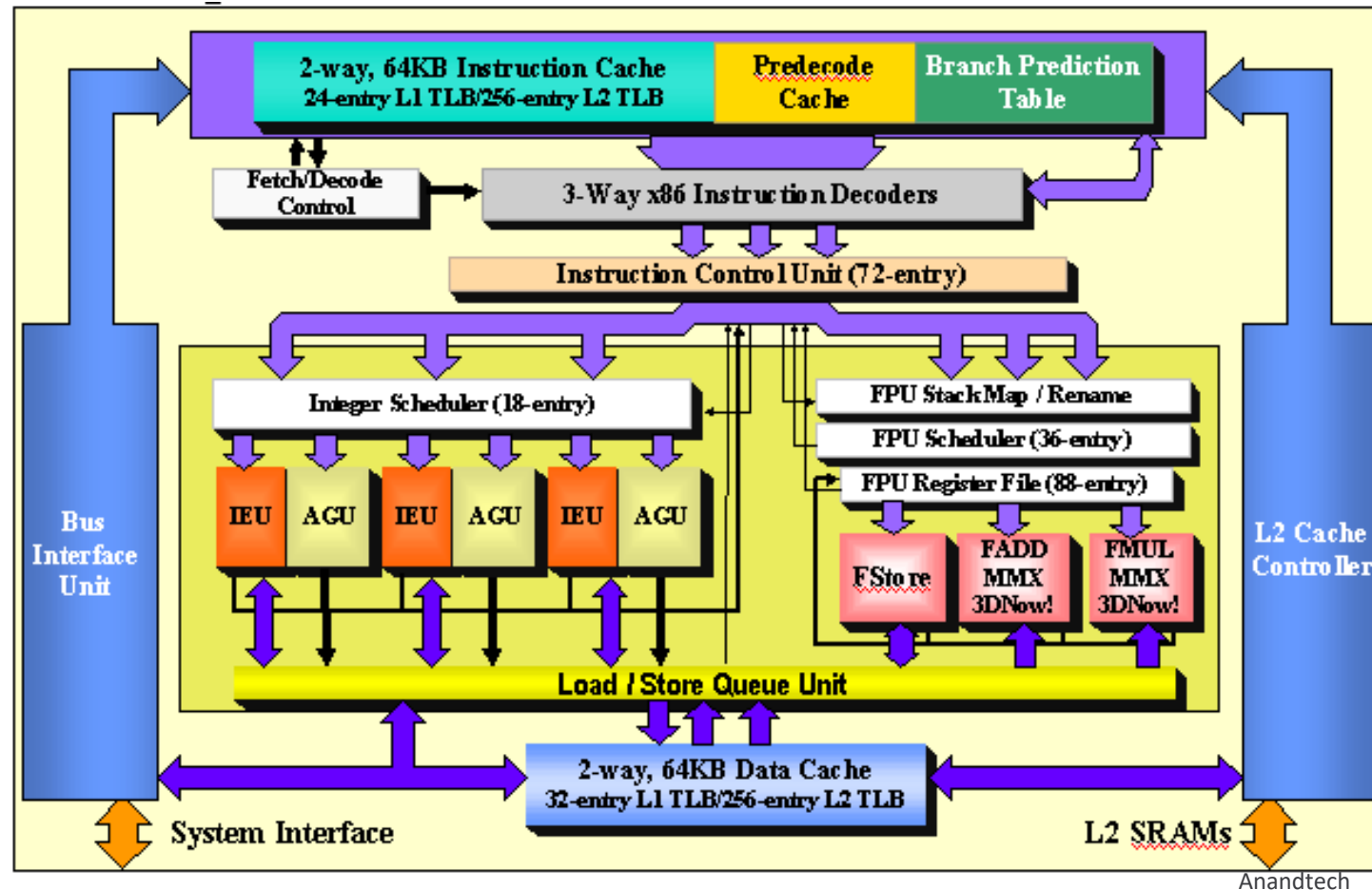
Instruction Set Architecture (ISA)

- **Contract between hardware and software**
- Hardware free to implement it in different ways
... as long as software can't tell the difference!
 - Improvements across processor generations
 - Design choices across product families (e.g., high-performance vs. low-energy)
 - High-performance trickery (e.g., out-of-order execution)
- Software free to use any syntax
... as long as it can be translated into working assembly program!

A simple implementation (made up)



A complex implementation (AMD Athlon)



ARM Cortex-M architecture

- 32-bit datapath (operands and results)
- 32-bit addressing space (memory size)
- *Thumb* ISA (vs. *ARM* ISA in Cortex-Ax)
 - Most instructions 16-bit encoding for compactness
 - Some instructions 32 bits to encode additional functionality
- Different products: M0, M0+, M3, M4, M7
 - “Core” ISA is the same; extensions for functionality
- ARM is *fabless*: License IP, implementation up to customer

Scalable and Compatible Architecture



ARM CORTEX
Processor Technology

Cortex-M0



Lowest cost
Low area



ARM CORTEX
Processor Technology

Cortex-M0+



Lowest power
Outstanding energy
efficiency



ARM CORTEX
Processor Technology

Cortex-M3



Performance efficiency
Feature rich connectivity



ARM CORTEX
Processor Technology

Cortex-M4



Digital Signal Control (DSC)
Processor with DSP
Accelerated SIMD
Floating point (FP)



ARM CORTEX
Processor Technology

Cortex-M7



Maximum DSC Performance
Flexible Memory System
Cache, TCM, AXI, ECC
Double & Single Precision FP

Digital Signal Control application space

'8/16-bit' Traditional application space

'16/32-bit' Traditional application space

ARM



Instruction set summary

| Instruction Type | Instructions |
|-------------------------|--|
| Move | MOV |
| Load/Store | LDR, LDRB, LDRH, LDRSH, LDRSB, LDM, STR, STRB, STRH, STM |
| Add, Subtract, Multiply | ADD, ADDS, ADCS, ADR, SUB, SUBS, SBCS, RSBS, MULS |
| Compare | CMP, CMN |
| Logical | ANDS, EORS, ORRS, BICS, MVNS, TST |
| Shift and Rotate | LSLS, LSRS, ASRS, RORS |
| Stack | PUSH, POP |
| Conditional branch | IT, B, BL, B{cond}, BX, BLX |
| Extend | SXTH, SXTB, UXTH, UXTB |
| Reverse | REV, REV16, REVSH |
| Processor State | SVC, CPSID, CPSIE, SETEND, BKPT |
| No Operation | NOP |
| Hint | SEV, WFE, WFI, YIELD |

ARM



Instruction format

- General format: `op <dst> <src1> <src2>`
 - There may be fewer source operands and/or no destination
 - Operands may be registers, or (sometimes) immediate constants
- Some examples:
 - `SUB R7,R2,R4` (“subtract”)
 - $R7 \leftarrow R2 - R4$
 - `SUBS R2,R2,#3` (“subtract and update status flags”)
 - $R2 \leftarrow R2 - 3$; update status flags (`SUBS` vs. `SUB`) according to result (will cover shortly)

Instruction format

- General format: `op <dst> <src1> <src2>`
 - There may be fewer source operands and/or no destination
 - Operands may be registers, or (sometimes) immediate constants
- Some more examples:
 - `CMP R2,R4` (“compare”)
 - Update status flags (will cover shortly) according to result of R2–R4; drop result
 - `BNE <label>` (“branch if not equal”)
 - Jump (branch) to instruction at position <label> in the program if status flag Z ≠ 0
 - Operand actually encoded as offset from current position in the program

Operands

- General-purpose registers
 - R0-R7 “low registers,” accessible by all instructions
 - R8-12 “high registers,” not accessible by many 16-bit instructions
 - R13-15 reserved for special purposes (will cover shortly)
 - R15 = “program counter” (PC); R14 = “link register” (LR); R13 = “stack pointer” (SP).
 - Write to at your own peril!
- Immediate values
 - Encoded within the instruction format
- Memory locations (will cover shortly)

Instruction encoding

- 32-bit instruction if bits [15:11] of the first half-word are 0x1d-f
 - Otherwise, 16-bit instruction
- Opcodes must be unambiguous
 - First few bits tell decoder what to expect in the rest of the instruction

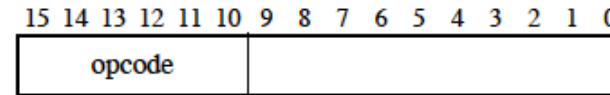


Table A5-1 shows the allocation of 16-bit instruction encodings.

Table A5-1 16-bit Thumb instruction encoding

| opcode | Instruction or instruction class |
|--------|--|
| 00xxxx | Shift (immediate), add, subtract, move, and compare on page A5-6 |
| 010000 | Data processing on page A5-7 |
| 010001 | Special data instructions and branch and exchange on page A5-8 |
| 01001x | Load from Literal Pool, see <i>LDR (literal)</i> on page A6-90 |
| 0101xx | Load/store single data item on page A5-9 |
| 011xxx | |
| 100xxx | |
| 10100x | Generate PC-relative address, see <i>ADR</i> on page A6-30 |
| 10101x | Generate SP-relative address, see <i>ADD (SP plus immediate)</i> on page A6-26 |
| 1011xx | Miscellaneous 16-bit instructions on page A5-10 |
| 11000x | Store multiple registers, see <i>STM / STMIA / STMEA</i> on page A6-218 |
| 11001x | Load multiple registers, see <i>LDM / LDMIA / LDMFD</i> on page A6-84 |
| 1101xx | Conditional branch, and supervisor call on page A5-12 |
| 11100x | Unconditional Branch, see <i>B</i> on page A6-40 |

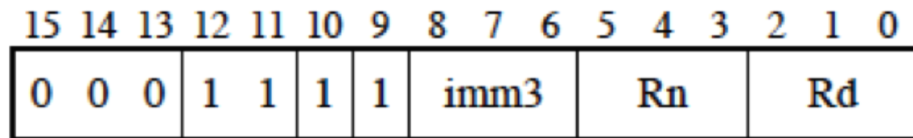
ARM

Example: **sub** (immediate)

Encoding T1 All versions of the Thumb ISA.

SUBS <Rd>, <Rn>, #<imm3>

SUB<C> <Rd>, <Rn>, #<imm3>

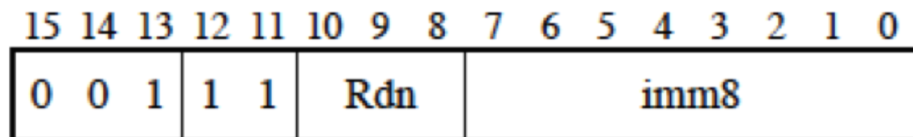


d = UInt(Rd); n = UInt(Rn); setflags = !InITBlock(); imm32 = ZeroExtend(imm3, 32);

Encoding T2 All versions of the Thumb ISA.

SUBS <Rdn>, #<imm8>

SUB<C> <Rdn>, #<imm8>



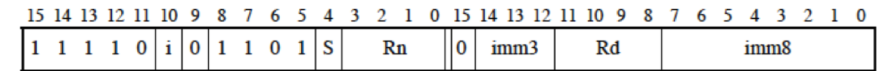
d = UInt(Rdn); n = UInt(Rdn); setflags = !InITBlock(); imm32 = ZeroExtend(imm8, 32);

Outside IT block.

Inside IT block.

Encoding T3 ARMv7-M

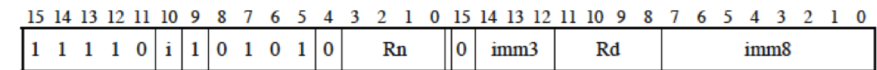
SUB{S}<C>.W <Rd>, <Rn>, #<const>



if Rd == '1111' && setflags then SEE CMP (immediate);
 if Rn == '1101' then SEE SUB (SP minus immediate);
 d = UInt(Rd); n = UInt(Rn); setflags = (S == '1'); imm32 = ThumbExpandImm(i:imm3:imm8);
 if d IN {13,15} || n == 15 then UNPREDICTABLE;

Encoding T4 ARMv7-M

SUBW<C> <Rd>, <Rn>, #<imm12>



if Rn == '1111' then SEE ADR;
 if Rn == '1101' then SEE SUB (SP minus immediate);
 d = UInt(Rd); n = UInt(Rn); setflags = FALSE; imm32 = ZeroExtend(i:imm3:imm8, 32);
 if d IN {13,15} then UNPREDICTABLE;

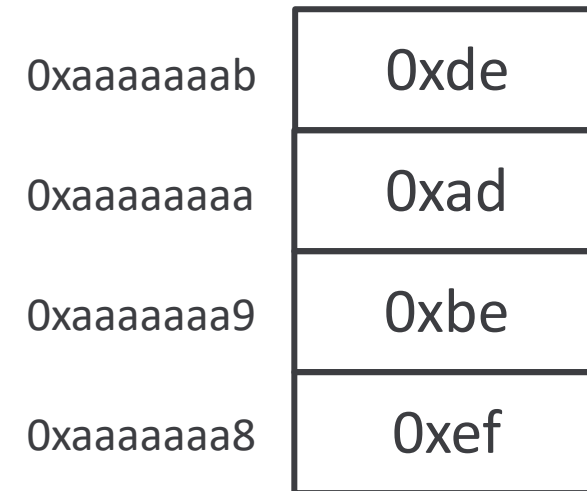
ARM

ARM



Memory organization

- Integer value types: byte (8b), half word (16b), word (32b)
- 32-bit addresses = 4 GB addressing space
 - Addressable by byte
- Words and half words *aligned*
 - E.g., 4-byte word \Rightarrow base address divisible by 4; value stored in locations $\text{base} + \{0, 1, 2, 3\}$
 - Cortex-M typ. *little-endian*: least-significant byte stored in base address (vs. most-significant byte in *big-endian*)
- Example: Write all legally accessible values \longrightarrow



Load/store operations

- ARM is a load-store architecture
 - Memory values can only be accessed through load/store instructions
 - All data processing takes place in registers
 - Dramatically reduces complexity of ISA and implementation
- **LDR** `<Rt>, <address>`: load (32-bit) word in M[address] into Rt
- **STR** `<Rt>, <address>`: store Rt's (32-bit) content into M[address]
- Other opcodes for half-word, byte, etc.

Loading sub-word data sizes

- How to load half-word/byte data into (32-bit) register?
 - Unsigned: Pad with zeroes—e.g., 0x82 (130) → 0x00000082
 - Signed: Sign extension—e.g., 0x82 (-126) → 0xffffffff82

| | Signed | Unsigned |
|-----------|--------|----------|
| Byte | LDRSB | LDRB |
| Half-word | LDRSH | LDRH |

ARM

- Can also sign-extend sub-word value already in a register:

| | Signed | Unsigned |
|-----------|--------|----------|
| Byte | SXTB | UXTB |
| Half-word | SXTH | UXTH |

ARM


Addressing modes

- Addressing modes: Calculate *effective address* on the fly
 - Few modes \implies simpler ISA and implementation
- [**<Rn>**, **<offset>**]: effective address is **<Rn>+<offset>**
 - **<Rn>** is the “base register;” it can be R0-7, PC, or SP
 - **<offset>** can be immediate constant or another register **<Rm>**
- [**<Rn>**, **<offset>**]!: Write effective address back to base register (“pre-update”)
- [**<Rn>**] , **<offset>**: Use base register as effective address, then update base register with newly calculated address (“post-update”)

Condition codes

- Special APSR register holds four one-bit *condition codes*
 - *Application Program Status Register*
 - N: Result of last status-updating instruction was Negative
 - Z: Result of last status-updating instruction was Zero
 - C: Last status-updating instruction produced Carry
 - V: Last status-updating instruction produced overflow
- “s” suffix indicates ALU instruction updates APSR
 - E.g., SUB vs. SUBS, ADC vs. ADCS, etc.
 - Compare instructions CMP, CMN always update APSR (obviously)

Branches

- Goal: change program flow
- Unconditional: **B <label>**
 - <label> limited to within ~2 kB of branch
- Conditional: **BXX <label>**
 - <label> limited to within ~256 B of branch
 - XX one of 

| Mnemonic extension | Meaning | Condition flags |
|------------------------|------------------------------|-------------------|
| EQ | Equal | Z == 1 |
| NE | Not equal | Z == 0 |
| CS ^a | Carry set | C == 1 |
| CC ^b | Carry clear | C == 0 |
| MI | Minus, negative | N == 1 |
| PL | Plus, positive or zero | N == 0 |
| VS | Overflow | V == 1 |
| VC | No overflow | V == 0 |
| HI | Unsigned higher | C == 1 and Z == 0 |
| LS | Unsigned lower or same | C == 0 or Z == 1 |
| GE | Signed greater than or equal | N == V |
| LT | Signed less than | N != V |
| GT | Signed greater than | Z == 0 and N == V |
| LE | Signed less than or equal | Z == 1 or N != V |
| None (AL) ^d | Always (unconditional) | Any |

ARM

Pseudo-instructions

- Assembly-like syntax, but emulated
 - Another instruction can accomplish the same
 - Small block of code (less frequent)
 - Part of the ISA specification; sometimes assembler-specific
- Example: `LDR <Rt>, <immediate>`
 - If <immediate> representable with 8 bits, use `MOV <Rt>, <immediate>`
 - Otherwise (one possible solution):
 - Place <immediate> in program's *literal pool* (well-known memory block)
 - Use `LDR <Rt>, [PC, <offset>]` where <offset> indicates position of literal relative to current PC

Example code: What does this do?

```
begin:                                     BEQ  end
    LDR  R1 ,=addr1                       CMP  R5 ,#0
    LDR  R2 ,=addr2                       BEQ  end
    LDR  R3 ,=addr3                       STR  R4 , [R3] , #4
next:                                     STR  R5 , [R3] , #4
    LDR  R4 , [R1] , #4                   B   next
    LDR  R5 , [R2] , #4                   end:
    CMP  R4 , #0                           WFI
```