Introduction to Assembly Language

ECE 3140/CS 3420 — EMBEDDED SYSTEMS
What is assembly language?

- **Assembly code**: Human-readable, quasi-isomorphc 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

Program counter: Keeps track of where processor is in the program

```
1 1 0 1 0 1 0 1 0 0
1 0 0 1 1 1 1 1 1 1
```

Intro to Assembly Language
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

---

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

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
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

**Assembly equivalent (made up)**

```
<table>
<thead>
<tr>
<th>op</th>
<th>src1/dst</th>
<th>src2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUB</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

R2 ← R2−R4; Z ← R2 == 0; PC ← PC+1

<table>
<thead>
<tr>
<th>op</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNZ</td>
<td>-1</td>
</tr>
</tbody>
</table>

PC ← 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)
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
ARM® CORTEX® Processor Technology

Cortex-M7

Scalable and Compatible Architecture

ARM® CORTEX® Processor Technology
Cortex-M0+

15 years

Lowest cost
Low area

Lowest power
Outstanding energy efficiency

Performance efficiency
Feature rich connectivity

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

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

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move</td>
<td>MOV</td>
</tr>
<tr>
<td>Load/Store</td>
<td>LDR, LDRB, LDRH, LDRSH, LDRSB, LDM, STR, STRB, STRH, STM</td>
</tr>
<tr>
<td>Add, Subtract, Multiply</td>
<td>ADD, ADDS, ADCS, ADR, SUB, SUBS, SBCS, RSBS, MULS</td>
</tr>
<tr>
<td>Compare</td>
<td>CMP, CMN</td>
</tr>
<tr>
<td>Logical</td>
<td>ANDS, EORS, ORRS, BICS, MVNS, TST</td>
</tr>
<tr>
<td>Shift and Rotate</td>
<td>LSLS, LSRS, ASRS, RORS</td>
</tr>
<tr>
<td>Stack</td>
<td>PUSH, POP</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>IT, B, BL, B{cond}, BX, BLX</td>
</tr>
<tr>
<td>Extend</td>
<td>SXTH, SXTB, UXTH, UXTB</td>
</tr>
<tr>
<td>Reverse</td>
<td>REV, REV16, REVSH</td>
</tr>
<tr>
<td>Processor State</td>
<td>SVC, CPSID, CPSIE, SETEND, BKPT</td>
</tr>
<tr>
<td>No Operation</td>
<td>NOP</td>
</tr>
<tr>
<td>Hint</td>
<td>SEV, WFE, WFI, YIELD</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>opcode</th>
<th>Instruction or instruction class</th>
</tr>
</thead>
<tbody>
<tr>
<td>00xxxx</td>
<td>Shift (immediate), add, subtract, move, and compare on page A5-6</td>
</tr>
<tr>
<td>010000</td>
<td>Data processing on page A5-7</td>
</tr>
<tr>
<td>010001</td>
<td>Special data instructions and branch and exchange on page A5-8</td>
</tr>
<tr>
<td>01001x</td>
<td>Load from Literal Pool, see LDR (literal) on page A6-90</td>
</tr>
<tr>
<td>0101xx</td>
<td>Load/store single data item on page A5-9</td>
</tr>
<tr>
<td>011xxx</td>
<td></td>
</tr>
<tr>
<td>100xxx</td>
<td></td>
</tr>
<tr>
<td>10100x</td>
<td>Generate PC-relative address, see ADR on page A6-30</td>
</tr>
<tr>
<td>10101x</td>
<td>Generate SP-relative address, see ADD (SP plus immediate) on page A6-26</td>
</tr>
<tr>
<td>1011xx</td>
<td>Miscellaneous 16-bit instructions on page A5-10</td>
</tr>
<tr>
<td>11000x</td>
<td>Store multiple registers, see STM / STMIA / STMEA on page A6-218</td>
</tr>
<tr>
<td>11001x</td>
<td>Load multiple registers, see LDM / LDMIA / LDMFD on page A6-84</td>
</tr>
<tr>
<td>1101xx</td>
<td>Conditional branch, and supervisor call on page A5-12</td>
</tr>
<tr>
<td>11100x</td>
<td>Unconditional Branch, see B on page A6-40</td>
</tr>
</tbody>
</table>

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

Table A5-1 16-bit Thumb instruction encoding
Example: \texttt{sub} (immediate)

\begin{itemize}
  \item \textbf{Encoding T1} \hspace{1cm} All versions of the Thumb ISA.
  \begin{equation}
    \text{SUBS} <Rd>,<Rn>,\#<imm3>
  \end{equation}
  \begin{equation}
    \begin{array}{cccccccccc}
      15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
      0 & 0 & 0 & 1 & 1 & 1 & 1 & \text{imm3} & Rn & Rd \\
    \end{array}
  \end{equation}
  
  \text{d} = \text{UInt} (Rd); \ n = \text{UInt} (Rn); \ \text{setflags} = \text{!InITBlock} (); \ \text{imm32} = \text{ZeroExtend} (\text{imm3}, 32);

  \item \textbf{Encoding T2} \hspace{1cm} All versions of the Thumb ISA.
  \begin{equation}
    \text{SUBS} <Rdn>,\#<imm8>
  \end{equation}
  \begin{equation}
    \begin{array}{cccccccccc}
      15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
      0 & 0 & 1 & 1 & 1 & 1 & \text{Rdn} & \text{imm8} \\
    \end{array}
  \end{equation}
  
  \text{d} = \text{UInt} (Rdn); \ n = \text{UInt} (Rdn); \ \text{setflags} = \text{!InITBlock} (); \ \text{imm32} = \text{ZeroExtend} (\text{imm8}, 32);
\end{itemize}
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 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

<table>
<thead>
<tr>
<th>0xaaaaaaaaab</th>
<th>0xaaaaaaa9</th>
<th>0xaaaaaaa8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xdec</td>
<td>0xad</td>
<td>0xbe</td>
</tr>
</tbody>
</table>
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.
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

<table>
<thead>
<tr>
<th></th>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Byte</strong></td>
<td>LDRSB</td>
<td>LDRB</td>
</tr>
<tr>
<td><strong>Half-word</strong></td>
<td>LDRSH</td>
<td>LDRH</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Byte</strong></td>
<td>SXTB</td>
<td>UXTB</td>
</tr>
<tr>
<td><strong>Half-word</strong></td>
<td>SXTH</td>
<td>UXTH</td>
</tr>
</tbody>
</table>
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:

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

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