Assembly Language

Readings: 2.1-2.7, 2.9-2.10, 2.14
   Green reference card

Assembly language
   Simple, regular instructions – building blocks of C, Java & other languages
   Typically one-to-one mapping to machine language

Our goal
   Understand the basics of assembly language
   Help figure out what the processor needs to be able to do

Not our goal to teach complete assembly/machine language programming
   Floating point
   Procedure calls
   Stacks & local variables
Aside: C/C++ Primer

```c
struct coord { int x, y; }; /* Declares a type */
struct coord start;         /* Object with two slots, x and y */
start.x = 1;                 /* For objects “.” accesses a slot */
struct coord *myLoc;         /* “*” is a pointer to objects */
myLoc = &start;              /* “&” returns thing’s location */
myLoc->y = 2;                /* “->” is “*” plus “.” */

int scores[8];              /* 8 ints, from 0..7 */
scores[1]=5;                 /* Access locations in array */
int *index = scores;         /* Points to scores[0] */
index++;                     /* Next scores location */
(*index)++;                  /* “*” works in arrays as well */
index = &(scores[3]);        /* Points to scores[3] */
*index = 9;
```

```
<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
```
ARM Assembly Language

The basic instructions have four components:

- Operator name
- Destination
- 1\textsuperscript{st} operand
- 2\textsuperscript{nd} operand

\begin{verbatim}
ADD <dst>, <src1>, <src2>     // <dst> = <src1> + <src2>
SUB <dst>, <src1>, <src2>     // <dst> = <src1> - <src2>
\end{verbatim}

Simple format: easy to implement in hardware

More complex: \( A = B + C + D - E \)
Operands & Storage

For speed, CPU has 32 general-purpose registers for storing most operands. For capacity, computer has large memory (multi-GB).

Load/store operation moves information between registers and main memory. All other operations work on registers.
## Registers

32x 64-bit registers for operands

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>X0-X7</td>
<td>Function arguments/Results</td>
<td></td>
</tr>
<tr>
<td>X8</td>
<td>Result, if a pointer</td>
<td></td>
</tr>
<tr>
<td>X9-X15</td>
<td>Volatile Temporaries</td>
<td>Not saved on call</td>
</tr>
<tr>
<td>X16-X17</td>
<td>Linker scratch registers</td>
<td>Don’t use them</td>
</tr>
<tr>
<td>X18</td>
<td>Platform register</td>
<td>Don’t use this</td>
</tr>
<tr>
<td>X19-X27</td>
<td>Temporaries (saved across calls)</td>
<td>Saved on call</td>
</tr>
<tr>
<td>X28</td>
<td>Stack Pointer</td>
<td></td>
</tr>
<tr>
<td>X29</td>
<td>Frame Pointer</td>
<td></td>
</tr>
<tr>
<td>X30</td>
<td>Return Address</td>
<td></td>
</tr>
<tr>
<td>X31</td>
<td>Always 0</td>
<td>No-op on write</td>
</tr>
</tbody>
</table>
Basic Operations

(Note: just subset of all instructions)

**Mathematic:** ADD, SUB, MUL, SDIV

*Immediate (one input a constant)*

```
ADD X0, X1, X2  // X0 = X1+X2
ADDI X0, X1, #100 // X0 = X1+100
```

**Logical:** AND, ORR, EOR

*Immediate*

```
AND X0, X1, X2  // X0 = X1&X2
ANDI X0, X1, #7  // X0 = X1&b0111
```

**Shift:** left & right logical (LSL, LSR)

```
LSL X0, X1, #4  // X0 = X1<<4
```

Example: Take bits 6-4 of X0 and make them bits 2-0 of X1, zeros otherwise:
Memory Organization

Viewed as a large, single-dimension array, with an address. A memory address is an index into the array. "Byte addressing" means that the index points to a byte of memory.
Memory Organization (cont.)

Bytes are nice, but most data items use larger units.

- Double-word = 64 bits = 8 bytes
- Word = 32 bits = 4 bytes

\[
\begin{array}{c|c}
0 & 64 \text{ bits of data} \\
8 & 64 \text{ bits of data} \\
16 & 64 \text{ bits of data} \\
24 & 64 \text{ bits of data} \\
\end{array}
\]

Registers hold 64 bits of data

\(2^{64}\) bytes with byte addresses from 0 to \(2^{64}-1\)

\(2^{61}\) double-words with byte addresses 0, 8, 16, ... \(2^{64}-8\)

Double-words and words are aligned

i.e., what are the least 3 significant bits of a double-word address?
Addressing Objects: Endian and Alignment

**Doubleword:**

<table>
<thead>
<tr>
<th></th>
<th>2^{256}..2^{56}</th>
<th>2^{248}..2^{54}</th>
<th>2^{240}..2^{52}</th>
<th>2^{232}..2^{80}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big Endian</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>2^{256}..2^{56}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>2^{248}..2^{54}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>2^{240}..2^{52}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>2^{232}..2^{80}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>2^{232}..2^{80}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>2^{216}..2^{88}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>2^{15}..2^{88}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>2^{8}..2^{8}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Little Endian |
| 000     | 2^{256}..2^{56} |
| 001     | 2^{248}..2^{54} |
| 010     | 2^{240}..2^{52} |
| 011     | 2^{232}..2^{80} |
| 100     | 2^{232}..2^{80} |
| 101     | 2^{216}..2^{88} |
| 110     | 2^{15}..2^{88}  |
| 111     | 2^{8}..2^{8}    |

Big Endian: address of most significant byte = doubleword address
Motorola 68k, MIPS, IBM 360/370, Xilinx Microblaze, Sparc

Little Endian: address of least significant byte = doubleword address
Intel x86, DEC Vax, Altera Nios II, Z80

ARM: can do either – this class assumes Little-Endian.
### Data Storage

**Characters:** 8 bits (byte)

**Integers:** 64 bits (D-word)

**Array:** Sequence of locations

**Pointer:** Address (64 bits)

```
// G = ASCII 71
char a = 'G';
int x = 258;
char *b;
int *y;
b = new char[4];
y = new int[10];
```

(Note: real compilers place local variables (the “stack”) from beginning of memory, new’ed structures (the “heap”) from end. We ignore that here for simplicity)
**Loads & Stores**

Loads & Stores move data between memory and registers

All operations on registers, but too small to hold all data

LDUR X0, [X1, #14] // X0 = Memory[X1+14]

STUR X2, [X3, #20] // Memory[X3+20] = X2

Note: LDURB & STURB load & store bytes
Addressing Example

The address of the start of a character array is stored in X0. Write assembly to load the following characters:

X2 = Array[0]

X3 = Array[1]

X4 = Array[2]

X5 = Array[k] // Assume the value of k is in X1
/* Swap the kth and (k+1)th element of an array */
swap(int v[], int k) {
    int temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}

// Assume v in X0, k in X1

// Array Example

[Table of GPRs with values 928, 10, empty, empty, and empty columns]

[Table of Memory with hexadecimal values: 0A12170D34BC2DE1, 1111111111111111, 0000000000000000, 0F0F0F0F0F0F0F0F, FFFFFFFFFFFFFFFFF]
Execution Cycle Example

PC: Program Counter
IR: Instruction Register

General Purpose Registers

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X0</td>
<td>928</td>
</tr>
<tr>
<td>X1</td>
<td>10</td>
</tr>
<tr>
<td>X2</td>
<td></td>
</tr>
<tr>
<td>X3</td>
<td></td>
</tr>
<tr>
<td>X4</td>
<td></td>
</tr>
</tbody>
</table>

Note: Word addresses Instructions are 32b

Instruction Fetch

Instruction Decode

Operand Fetch

Execute

Result Store

Next Instruction

Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>D3600C22</td>
</tr>
<tr>
<td>0004</td>
<td>8B020002</td>
</tr>
<tr>
<td>0008</td>
<td>F8400043</td>
</tr>
<tr>
<td>0012</td>
<td>F8408044</td>
</tr>
<tr>
<td>0016</td>
<td>F8400044</td>
</tr>
<tr>
<td>0020</td>
<td>F8408043</td>
</tr>
<tr>
<td>1000</td>
<td>OA12170D34BC2DE1</td>
</tr>
<tr>
<td>1008</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>1016</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>1024</td>
<td>0F0F0F0F0F0F0F0F</td>
</tr>
<tr>
<td>1032</td>
<td>FFFFFFFFFFFFFFFF</td>
</tr>
<tr>
<td>1040</td>
<td>FFFFFFFFFFFFFFFF</td>
</tr>
</tbody>
</table>
Flags/Condition Codes

Flag register holds information about result of recent math operation
   Negative: was result a negative number?
   Zero: was result 0?
   Overflow: was result magnitude too big to fit into 64-bit register?
   Carry: was the carry-out true?

Operations that set the flag register contents:
   ADDS, ADDIS, ANDS, ANDIS, SUBS, SUBIS, some floating point.

Most commonly used are subtracts, so we have a synonym: CMP
   CMP X0, X1  same as SUBS X31, X0, X1
   CMPI X0, #15 same as SUBIS X31, X0, #15
Control Flow

Unconditional Branch – GOTO different next instruction

- B START // go to instruction labeled with “START” label
- BR X30 // go to address in X30: PC = value of X30

Conditional Branches – GOTO different next instruction if condition is true

1 register: CBZ (==0), CBNZ (!= 0)

- CBZ X0, FOO // if X0 == 0 GOTO FOO: PC = Address of instr w/FOO label

2 register: B.LT (<), B.LE(<=), B.GE (>=), B.GT(>), B.EQ(==), B.NE(!=)

- first compare (CMP X0, X1, CMPI X0, #12), then b.cond instruction
- CMP X0, X1 // compare X0 with X1 – same as SUBS X31, X0, X1
- B.EQ FOO // if X0 == X1 GOTO FOO: PC = Address of instr w/FOO label

if (a == b)
    a = a + 3;
else
    b = b + 7;
c = a + b;

// X0 = a, X1 = b, X2 = c
- CMP X0, X1 // set flags
- B.NE ELSEIF // branch if a!=b
- ADDI X0, X0, #3 // a = a + 3
- B DONE // avoid else
ELSERIF:
    ADDI X1, X1, #7 // b = b + 7
DONE:
    ADD, X2, X0, X1 // c = a + b
Loop Example

Compute the sum of the values 0…N-1

```java
int sum = 0;
for (int I = 0; I != N; I++) {
    sum += I;
}

// X0 = N, X1 = sum, X2 = I
```
String toUpper

Convert a string to all upper case

```c
char *index = string;
while (*index != 0) { /* C strings end in 0 */
    if (*index >= ‘a’ && *index <= ‘z’)
        *index = *index +(‘A’ - ‘a’);
    index++;
}

// string is a pointer held at Memory[80].
// X0=index, ‘A’ = 65, ‘a’ = 97, ‘z’ = 122
```
# Machine Language vs. Assembly Language

<table>
<thead>
<tr>
<th>Assembly Language</th>
<th>Machine language</th>
</tr>
</thead>
<tbody>
<tr>
<td>mnemonics for easy reading</td>
<td>Completely numeric representation</td>
</tr>
<tr>
<td>labels instead of fixed addresses</td>
<td>format CPU actually uses</td>
</tr>
<tr>
<td>Easier for programmers</td>
<td></td>
</tr>
<tr>
<td>Almost 1-to-1 with machine language</td>
<td></td>
</tr>
</tbody>
</table>

**SWAP:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
<th>Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSL X9, X1, #3</td>
<td>// Compute address of v[k]</td>
<td>11010011011 00000 000011 00001 01001</td>
</tr>
<tr>
<td>ADD X9, X0, X9</td>
<td>// get v[k]</td>
<td>10001011000 01001 00000 00000 01001</td>
</tr>
<tr>
<td>LDUR X10, [X9, #0]</td>
<td>// get v[k+1]</td>
<td>11111000010 000000000 00 01001 01010</td>
</tr>
<tr>
<td>LDUR X11, [X9, #8]</td>
<td>// save new value to v[k]</td>
<td>11110000010 000001000 00 01001 01011</td>
</tr>
<tr>
<td>STUR X11, [X9, #0]</td>
<td>// save new value to v[k+1]</td>
<td>11110000000 000000000 00 01001 01011</td>
</tr>
<tr>
<td>STUR X10, [X9, #8]</td>
<td>// return from subroutine</td>
<td>11110000000 000001000 00 01001 01010</td>
</tr>
<tr>
<td>BR X30</td>
<td></td>
<td>11010110000 00000 00000 00000 11110</td>
</tr>
</tbody>
</table>
Labels

Labels specify the address of the corresponding instruction
Programmer doesn’t have to count line numbers
Insertion of instructions doesn’t require changing entire code

```plaintext
// X0 = N, X1 = sum, X2 = I
ADD X1, X31, X31 // sum = 0
ADD X2, X31, X31 // I = 0

TOP:
   CMP X2, X0 // Check I vs N
   B.GE END // end when !(I<N)
   ADD X1, X1, X2 // sum += I
   ADDI X2, X2, #1 // I++
   B TOP // next iteration

END:
```

Notes:

Branches are PC-relative

PC = PC + 4*(BranchOffset)
BranchOffset positive -> branch downward. Negative -> branch upward.
Labels Example

Compute the value of the labels in the code below.
Branches: PC = PC + 4*(BranchOffset)

// Program starts at address 100
LDUR   X0, [X31, #100]
LOOP:
   LDURB X1, [X0, #0]
   CBZ   X1, END
   CMPI  X1, #97
   B.LT  NEXT
   CMPI  X1, #122
   B.GT  NEXT
   SUBI  X1, X1, #32
   STURB X1, [X0, #0]
NEXT:
   ADDI  X0, X0, 1
   B     LOOP
END:
Instruction Types

Can group instructions by # of operands

3-register

ADD X0, X1, X2
ADDI X0, X1, #100
AND X0, X1, X2
ANDI X0, X1, #7
LSL X0, X1, #4
LSR X0, X1, #2
LDUR X0, [X1, #14]

2-register

LDURB X0, [X1, #14]
STUR X0, [X1, #14]
STURB X0, [X1, #14]
B START
BR X30

1-register

CBZ X0, FOO
B.EQ DEST

0-register
## Instruction Formats

All instructions encoded in 32 bits (operation + operands/immediates)

**Branch (B-Type)**

- Instr[31:21] = 0A0-0BF
- **Opcode** | **BrAddr26**

**Conditional Branch (CB-Type)**

- Instr[31:21] = 2A0-2A7, 5A0-5AF
- **Opcode** | **CondAddr19** | **Rd**

**Register (R-Type)**

- **Opcode** | **Rm** | **SHAMT** | **Rn** | **Rd**

**Immediate (I-Type)**

- **Opcode** | **ALU_Imm12** | **Rn** | **Rd**

**Memory (D-Type)**

- Instr[31:21] = 1C0-1C2, 7C0-7C2
- **Opcode** | **DT_Address9** | **00** | **Rn** | **Rd**
B-Type

Used for unconditional branches

0x05: B

\[ \text{B} - 3 \quad // \quad \text{PC} = \text{PC} + 4\times -3 \]
CB-Type

Used for conditional branches

<table>
<thead>
<tr>
<th>Opcode</th>
<th>CondAddr19</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x54:</td>
<td>B.cond</td>
<td></td>
</tr>
<tr>
<td>0xB4:</td>
<td>CBZ</td>
<td></td>
</tr>
<tr>
<td>0xB5:</td>
<td>CBNZ</td>
<td></td>
</tr>
</tbody>
</table>

Condition Codes

- 0x00: EQ (==)
- 0x01: NE (!=)
- 0x0A: GE (>=)
- 0x0B: LT (<)
- 0x0C: GT (>
- 0x0D: LE (<=)

CBZ X12, -3  // if (X12==0) PC = PC + 4*-3

B.LT -5     // if (lessThan) PC = PC + 4*-5
# R-Type

Used for 3 register ALU operations and shift

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rm</th>
<th>SHAMT</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x450:</td>
<td>AND</td>
<td>Op2</td>
<td>Op1</td>
<td>Dest</td>
</tr>
<tr>
<td>0x458:</td>
<td>ADD</td>
<td>Shift amount</td>
<td>(0 for non-shift)</td>
<td>(0 for shift)</td>
</tr>
<tr>
<td>0x4D6:</td>
<td>SDIV, shamt=02</td>
<td>ADD</td>
<td>X3, X5, X6</td>
<td>// X3 = X5+X6</td>
</tr>
<tr>
<td>0x4D8:</td>
<td>MUL, shamt=1F</td>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x550:</td>
<td>ORR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x558:</td>
<td>ADDS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x650:</td>
<td>EOR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x658:</td>
<td>SUB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x69A:</td>
<td>LSR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x69B:</td>
<td>LSL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x6B0:</td>
<td>BR, rest all 0’s but Rd</td>
<td>LSL</td>
<td>X10, X4, #6</td>
<td>// X10 = X4&lt;&lt;6</td>
</tr>
<tr>
<td>0x750:</td>
<td>ANDS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x758:</td>
<td>SUBS</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## I-Type

**Used for 2 register & 1 constant ALU operations**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>ALU_Imm12</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x244:</td>
<td>ADDI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x248:</td>
<td>ANDI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x164:</td>
<td>ADDIS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x168:</td>
<td>ORRI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x344:</td>
<td>SUBI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x348:</td>
<td>EORI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x2C4:</td>
<td>SUBIS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x2C8:</td>
<td>ANDIS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Example:**

```plaintext
ADDI  X8, X3, #35 // X8 = X3+35
```

**Diagram:**

![Instruction Format Diagram](image-url)
# D-Type

**Used for memory accesses**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>DAddr9</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1C0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x1C2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7C0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7C2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Example:**

```assembly
LDUR X6, [X15, #12]  // X6 = Memory[X15+12]
```

<table>
<thead>
<tr>
<th>Address Constant</th>
<th>Address Reg</th>
<th>Target Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>DAddr9</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1C0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x1C2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7C0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7C2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[Diagram of D-Type instruction encoding]

---

36
Conversion example

Compute the sum of the values 0…N-1

```
ADD X1, X31, X31
ADD X2, X31, X31
B TEST
TOP:
  ADD X1, X1, X2
ADDI X2, X2, #1
TEST:
  SUBS X31, X2, X0
B.LT TOP
END:
```
Assembly & Machine Language

Assembly

Machine Language