Overview

- What computers really do
  - fetch / decode / execute cycle
- Assembly: action => to bits
- Decoding: bits => actions
- Disassembly
- Conclusion

Review: What is Subject about?

- Coordination of many levels of abstraction

Review: Programming Levels of Representation

- High Level Language Program (e.g., C)
- Assembler
- Machine Language Program (ARM)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
1110 0101 1001 0010 0000 0000 0000 0000
1110 0101 1001 0010 0000 0000 0000 0100
1110 0101 1000 0010 0001 0000 0000 0000
1110 0101 1000 0010 0001 0000 0000 0100
```

```
ldr r0 , [r2, #0]
ldr r1 , [r2, #4]
str r1 , [r2, #0]
str r0 , [r2, #4]
```

```
1110 0101 1000 0010 0001 0000 0000 0000
1110 0101 1001 0010 0000 0000 0000 0000
1110 0101 1001 0010 0000 0000 0000 0000
1110 0101 1001 0010 0000 0000 0000 0000
```

Review: What Does a Computer Do?

- **Big Idea: Stored Program Concept**
  - encode instructions as numbers, data as numbers, store them all in memory
  - Everything has an address

- **PC = address of current instruction to execute**

- **Fetch instruction at PC**

- **Decode it**

- **Do what it tells you to do**
  - updates registers and memory
  - updates PC

Review: What happens after ifetch/decode

- Perform the operations that are specified in the instruction
  - operand fetch: read values from registers
  - execute
    - perform arithmetic/logic operation => reg
    - perform ldr (mem => reg)
    - perform str (reg => mem)
  - compute next
    - PC <= PC + 4 for all of the above
    - PC <= jump, branch (if taken)

- then fetch/decode the next instruction

Fetch/Decode/Execute Cycle

- Inst <= Fetch MEM[ PC ]
- Decode( inst )
- case ARITH/LOG
  - REG<sub>inst</sub> <= REG<sub>inst</sub>, OP<sub>inst</sub>, REG<sub>inst</sub>
  - PC <= PC + 4
- case ARITH/LOG-immned
  - REG<sub>inst</sub> <= REG<sub>inst</sub>, OP<sub>inst</sub>, IM<sub>inst</sub>
  - PC <= PC + 4
- case LOAD
  - REG<sub>inst</sub> <= MEM[ REG<sub>inst</sub> + IM<sub>inst</sub> ]
  - PC <= PC + 4
- case STORE
  - MEM[ REG<sub>inst</sub> + IM<sub>inst</sub> ] <= REG<sub>inst</sub>
  - PC <= PC + 4
- case CONTROL
  - PC <= OP<sub>inst</sub>(PC, REG<sub>inst</sub>, IM<sub>inst</sub>)

Review: Instruction Set (ARM 7TDMI)

- Set of instruction that a processor can execute

- Instruction Categories
  - Data Processing or Computational (Logical and Arithmetic)
  - Load/Store (Memory Access)
  - Control Flow (Jump and Branch)
  - Floating Point
    - coprocessor
  - Memory Management
  - Special

- Special Register

- CPSR

- N Z C V
- unused
- IF T mode
ARM Data Processing Instructions

All instructions 32 bits wide

Immediate

Register & Imm. Shifted Register

Register Shifted Register

3 types of addressing modes

add r4, r5, #25
r4 ← r5 + 25

add r4, r5, r6
r4 ← r5 + r6

add r4, r5, r6, lsl #2
r4 ← r5 + (r6 × 2²)

Register

Imm. Shifted Register

Immediate

ARM Load/Store Instructions (#1/3)

All instructions 32 bits wide

Immediate

Imm. Shifted Register

3 types of addressing modes

str r4, [r5, r6, lsl #2]
r4 → mem[r5 + (r6 × 2²)]

ARM Load/Store Instructions (#2/3)

All instructions 32 bits wide

Immediate preindexed

Imm. Shifted Register preindexed

3 types of addressing modes

ldr r4, [r5, #25]!
r4 ← mem[r5 + 25]
r5 ← r5 + 25

ARM Load/Store Instructions (#3/3)

All instructions 32 bits wide

Immediate post indexed

Imm. Shifted Register post indexed

3 types of addressing modes

str r4, [r5, r6, lsl #2]
r4 → mem[r5 + (r6 × 2²)]
r5 ← r5 + r6 × 2²
ARM Branch Instructions

All instructions 32 bits wide

PC = PC + (SignExt(24 offset) <<00)

Unconditional and Conditional Branches (L=0)

here: b there...

. . .

There: movs r4, r5

beq here

Branch & Link (L=1)

along with jump, the address of the next instruction is stored in r14.

go back to instruction after bl instruction

PC Relative Addressing

Conditional Execution Field

0000 = EQ - Z set (equal)
0001 = NE - Z clear (not equal)
0010 = HS / CS - C set (unsigned higher or same)
0011 = LO / CC - C clear (unsigned lower)
0100 = MI -N set (negative)
0101 = PL- N clear (positive or zero)
0110 = VS - V  set (overflow)
0111 = VC - V clear (no overflow)
1000 = HI - C set and Z clear (unsigned higher)
1001 = LS - C clear or Z set (unsigned lower or same)
1010 = GE - N set and V set, or N clear and V clear (signed >=)
1011 = LT - N set and V set, or N clear and V set (signed <)
1100 = GT - Z clear, and either N set and V set, or N clear and V clear (signed >)
1101 = LE - Z set, or N set and V clear, or N clear and V set (signed <, or =)
1110 = AL - always
1111 = NV - reserved.

Comparison: 

cmp r1, r2

Instr<cond> ---

31 28 27 26 25 24 23 20 16 15 8 7 6 5 4 3 2 1 0

Instruction type

Data processing / PSR transfer
Multiply
Long Multiply  (v3M / v4 only)
Swap
Load/Store Byte/Word
Load/Store Multiple
Halfword transfer:Immediate offset (v4 only)
Halfword transfer: Register offset (v4 only)
Branch
Branch Exchange  (v4T only)
Coprocessor data transfer
Coprocessor data operation
Coprocessor register transfer
Software interrupt

Reading Material


chapter A2: Programmer’s Model
Recall: Sample Assembly Program

C statement: \( k = k - 2 \)

Binary Contents

<table>
<thead>
<tr>
<th>r2</th>
<th>0x94</th>
<th>0x94</th>
</tr>
</thead>
<tbody>
<tr>
<td>r5</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>r2</td>
<td>0x94</td>
<td></td>
</tr>
<tr>
<td>r0</td>
<td>0x20</td>
<td></td>
</tr>
<tr>
<td>r0</td>
<td>0x20</td>
<td></td>
</tr>
<tr>
<td>r5</td>
<td>0x2</td>
<td></td>
</tr>
<tr>
<td>r0</td>
<td>0x1E</td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td>0x94</td>
<td></td>
</tr>
<tr>
<td>r0</td>
<td>0x1E</td>
<td></td>
</tr>
</tbody>
</table>

Location for variable \( k \)

Decoding Machine Language

How do we convert 1s and 0s to C code?

For each 32 bits:

- Look at bits 27 - 25: 00x means data processing, 01x Load/Store, 101 Branch.
- Use instruction type to determine which fields exist and convert each field into the decimal equivalent.
- Once we have decimal values, write out ARM assembly code.
- Logically convert this ARM code into valid C code.

Compilation & Assembly

How to turn notation programmers prefer into notation computer understands?

Program to translate C statements into Assembly Language instructions; called a compiler

- Example: compile by hand this C code:
  
  \[
  a = b + c; \\
  d = a - e;
  \]

  
  \[
  \text{Ass: } \quad \begin{align*}
  &\text{add r0, r1, r2} \\
  &\text{sub r3, r0, r4}
  \end{align*}
  \]

  
  \[
  \text{Big Idea: compiler translates notation from 1 level of abstraction to lower level}
  \]

Program to translate Assembly Language into machine instructions; called an assembler

- Ass: \( \begin{align*}
  &\text{add r0, r1, r2} \\
  &\text{sub r3, r0, r4}
  \end{align*} \)

- Mach: \( 0xe0810002, 0xe0403004 \)

- Big Idea: assembler translates notation from 1 level of abstraction to lower level

Decoding Example (#1/7)

Here are seven machine language instructions in hex:

\[
\begin{align*}
  &\text{e3520000} \\
  &\text{e3a00000} \\
  &\text{d1a0f00e} \\
  &\text{e2522001} \\
  &\text{e0800001} \\
  &\text{d1a0f00e} \\
  &\text{eaf5f5fb}
\end{align*}
\]

Let the first instruction be at address \( 4,194,304_{10} (0x00400000) \).

Next step: convert to binary
Decoding Example (#2/7)

- **Binary ⇒ Decimal ⇒ Assembly ⇒ C?**
- **Start at program at address 4,194,304₁₀ = 0x00400000 (2²²)**

```
11100011010100100000000000000000
111000111010000000000000000000001
110100110100001111000000001110
11010010100100000000000000000001
11000010000000000000000000000001
110100110100001111000000001110
111010101011111111111111111111011
```

- **What are instruction formats of these 7 instructions?**

Decoding Example (#3/7)

```
31 - 28 27-25 24 - 21 20-19 18-16 15-12 11-8 7 - 0
```

Decoding Example (#4/7)

```
14  1  10 1  2   0   0    0
14  1  13 0  0   0   0    0
13  0  13 0  0  15   0  00  14
14  1   2 1  2   2   0    1
14  0   4 0  0   0   0  00   1
13  0  13 0  0  15   0  00  14
14  5 0             -5
```

Decoding Example (#5/7)

```
4194304: cmp     r2, #0
4194308: mov     r0, #0
4194312: subs    r2, r2, #1
4194320: add     r0, r0, r1
4194324: movle   r15, r14
4194328: b       4194316 ;(pc–4*5)
```
Decoding Example (#6/7)

Binary ⇒ Fields ⇒ Decimal ⇒ Assembly ⇒
Symbolic Assembly ⇒ C?

Binary:
- 4194304: cmp r2, #0
- 4194308: mov r0, #0
- 4194312: movle r15, r14
- 4194316: subs r2, r2, #1
- 4194320: add r0, r0, r1
- 4194324: movle r15, r14

Assembly:
- 4194328: b 4194316

Symbolic Assembly:
- cmp r2, #0
- mov r0, #0
- movle r15, r14
- subs r2, r2, #1
- add r0, r0, r1
- movle r15, r14
- b Loop

Decoding Example (#7/7)

Binary ⇒ Fields ⇒ Decimal ⇒ Assembly ⇒
Symbolic Assembly ⇒ C?

Binary:
- A: cmp a3, #0
- R: mov a1, #0
- M: movle pc, lr

Assembly:
- Loop: subs a3, a3, #1
- add a1, a1, a2
- movle pc, lr
- b Loop

Mapping:
- product: a1, mcand: a2, mlier: a3

C

```
cmp r2, #0
mov r0, #0
movle r15, r14
Loop: subs r2, r2, #1
add r0, r0, r1
movle r15, r14
b Loop
```

```
cmp a3, #0
mov a1, #0
movle pc, lr
Loop: subs a3, a3, #1
add a1, a1, a2
movle pc, lr
b Loop
```

```
product = 0;
while (0 < mlier) {
    product += mcand;
    mlier -= 1;
}
```

Instruction Set Bridge

- more than 1-1 encode/decode
- many encoders & many decoders

```
C
Fortran
Java
ARM Assembly Language
GCC
CC
ARM-elf/ARM-AXD
ARM-elf
ARM Machine Language
```

Many different implementations of ARM Machine Language (Instruction Set)

“And in Conclusion…”

- Big Idea: fetch-decode-execute cycle
- Big Idea: encoding / decoding
- compiler/assembler encodes instructions as numbers, computer decodes and executes them
- keyboard encodes characters as numbers, decoded on display

- Instruction format
- certain fields determine how to decode the others
- each field has specific “decoding table” giving meaning to values
- highly structured and regular process