Overview

° What computers really do
  • fetch / decode / execute cycle
° Assembly: action $\Rightarrow$ to bits
° Decoding: bits $\Rightarrow$ 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)

Compiler

Assembly Language Program (e.g. ARM)

Assembler

Machine Language Program (ARM)

Machine Interpretation

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

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

1110 0101 1001 0010 0000 0000 0000 0100
1110 0101 1000 0010 0001 0000 0000 0000
1110 0101 1000 0010 0001 0000 0000 0100
1110 0101 1000 0010 0001 0000 0000 0100
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 \( \mathbf{\rightarrow} \) reg
    - perform ld (mem \( \mathbf{\rightarrow} \) reg)
    - perform st (reg \( \mathbf{\rightarrow} \) mem)
  - compute next
    - PC \( \leftarrow \) PC + 4 for all of the above
    - PC \( \leftarrow \) jump, branch (if taken)
- then fetch/decode the next instruction

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

Fetch/Decode/Execute Cycle

- Inst \( \leftarrow \) Fetch MEM[ PC ]
- Decode( inst ) case ARITH/LOG
  - REG\text{\_inst} \( \leftarrow \) REG\text{\_inst} OP\text{\_inst} REG\text{\_inst}
  - PC \( \leftarrow \) PC + 4
- case ARITH/LOG-immed
  - REG\text{\_inst} \( \leftarrow \) REG\text{\_inst} OP\text{\_inst} IM\text{\_inst}
  - PC \( \leftarrow \) PC + 4
- case LOAD
  - REG\text{\_inst} \( \leftarrow \) MEM[ REG\text{\_inst} + IM\text{\_inst} ]
  - PC \( \leftarrow \) PC + 4
- case STORE
  - MEM[ REG\text{\_inst} + IM\text{\_inst} ] \( \leftarrow \) REG\text{\_inst}
  - PC \( \leftarrow \) PC + 4
- case CONTROL
  - PC \( \leftarrow \) OP\text{\_inst}(PC, REG\text{\_inst}, IM\text{\_inst})
ARM Data Processing Instructions

All instructions 32 bits wide

Immediate

Register & Imm. Shifted Register

Register Shifted Register

3 types of addressing modes

ARM Load/Store Instructions (#1/3)

Add r4, r5, #25
r4 ← r5 + 25

Immediate

Immediate

Register Shifted Register

3 types of addressing modes

ARM Load/Store Instructions (#2/3)

Add r4, r5, r6
r4 ← r5 + r6

Immediate preindexed

Immediate

Imm. Shifted Register preindexed

3 types of addressing modes

ARM Load/Store Instructions (#3/3)

Add r4, r5, lsl #2
r4 ← r5 + (r6 × 2^2)

Immediate post indexed

Imm. Shifted Register post indexed

3 types of addressing modes
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

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 clear, 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.

cmp r1, r2
Instr<cond> --

31 28 27 25 24 23 20 16 12 8 4 0

Conditional Execution Field

唆

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

5 Rules that Comp Engineers Live by (#1/5)

- Engineered laws in between discovering the electron and putting 50 million transistors on an integrated circuit:

1. **Mother of A laws: Moore’s Law**
   
   Suggested by Intel Corp. legend Gordon E. Moore 38 years ago.
   
   *The number of transistors on a chip doubles annually.*
   
   The current growth rate is doubling/2Yrs
   
   Intel PR quotes doubling/18 months

2. **Rock’s Law**
   
   Suggested by Intel Corp. investor Arthur Rock
   
   *The cost of semiconductor tools doubles every four years*.
   
   If true it should have costed $5 billion a piece by the late 1990s and $10 billion by now.
   
   Not so. fabs cost $2 billion apiece, the same as in the late 1990s
   
   In addition productivity has gone up.
   
   In 80s fabs increased their yield;
   
   From 90s, fabs are increasing their throughput from 20 per hour in the early 90s to about 40 to 50 an hour today.
   
   Transistors have gone from a dime a dozen to a buck for a hundred billion (no lie).

3. **MACHRONE’S Law**
   
   Suggested by Bill Machrone, a long-time columnist for PC Magazine (1984)
   
   *The PC you want to buy will always be $5000*
   
   The magic number dropped to around $3000 in the early 1990s and held there until about 2000,
   
   Now an okay machine costs around $1500, although a fully loaded one will still run $5000."

4. **METCALFE’S Law**
   
   Suggested by Metcalfe, the inventor of the Ethernet standard and founder of the networking company 3Com Corp., (1980)
   
   *A network’s value grows proportionately to the number of its users squared*
   
   Telephone Example
   
   Hard to quantify
   
   Saturation
   
   Cacophony and clustering
   
   Network contaminants
5 Rules that Comp Engineers Live by (#5/5)

5. WIRTH’S Law

Suggested in 1995 by Niklaus Wirth of ETH Switzerland, inventor of the Pascal computer language,

Software is slowing faster than hardware is accelerating

“Groves giveth, and Gates taketh away.”

Andy Grove is another legend from Intel

Text editors of the early 1970s worked with 8000 bytes of storage, whereas modern equivalents demand 10000 times as much.

Useless Features: In Word 2000 you can spell “Greek” in Greek letters: \( \wp \sum \beta \).

Users tolerate “feature bloat” for reasons:
1. Moore’s Law, which makes the bloat possible,
2. Ignorance among consumers

The root cause is the interests of software companies

Source: IEEE Spectrum Dec 2003

Recall: Sample Assembly Program

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

Binary Contents

Instruction (Data proc.)

<table>
<thead>
<tr>
<th>Instruction (Data proc.)</th>
<th>Binary Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov r2, #0x94</td>
<td>E3A02094</td>
</tr>
<tr>
<td>mov r5, #2</td>
<td>E3A05002</td>
</tr>
<tr>
<td>ldr r0, [r2]</td>
<td>E5920000</td>
</tr>
<tr>
<td>sub r0, r0, r5</td>
<td>E0400005</td>
</tr>
<tr>
<td>str r0, [r2]</td>
<td>E5820000</td>
</tr>
</tbody>
</table>

Location for variable \( k \)

Data \( 20_{\text{hex}} = 32_{10} \)

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:
    \[ a = b + c; \]
    \[ d = a - e; \]
  • Ass:
    add r0, r1, r2
    sub r3, r0, r4
  • 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:
    add r0, r1, r2
    sub r3, r0, r4
  • Mach:
    0xe0810002
    0xe0403004
  • Big Idea: assembler translates notation from 1 level of abstraction to lower level

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, 10x 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.
Decoding Example (#1/7)

- Here are seven machine language instructions in hex:
  - e3520000
  - e3a00000
d1a0f00e
e2522001
e0800001
d1a0f00e
eafffffb
  
  Let the first instruction be at address $4,194,304 = 0x00400000$.

- Next step: convert to binary

Decoding Example (#2/7)

- Binary $\Rightarrow$ Decimal $\Rightarrow$ Assembly $\Rightarrow$ C?

- Start at program at address $4,194,304 = 0x00400000$ ($2^{22}$)
  - $11100011010100100000000000000000$ $11100011101000000000000000000000$ $11010001010010001000000000000001$ $11100000100000000000000000000001$ $110100011010000001111000000000110$ $1110101010111111111111111111111011$

- What are instruction formats of these 7 instructions?

Decoding Example (#3/7)

- Binary $\Rightarrow$ Fields $\Rightarrow$ Decimal $\Rightarrow$ Assembly $\Rightarrow$ C?

- DPI/DPR = Data Proc. immd./reg 2nd opernd
- CDPR = Cond. DPR
- BR = Branch

Decoding Example (#4/7)

- Binary $\Rightarrow$ Fields $\Rightarrow$ Decimal $\Rightarrow$ Assembly $\Rightarrow$ C?

- DPI/DPR = Data Proc. immd./reg 2nd opernd
- CDPR = Cond. DPR
- BR = Branch
### Decoding Example (#5/7)

<table>
<thead>
<tr>
<th>Binary</th>
<th>Fields</th>
<th>Decimal</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPI</td>
<td>14 1</td>
<td>10 1 2</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>DPI</td>
<td>14 1</td>
<td>13 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>CDPR</td>
<td>13 0</td>
<td>13 0 0</td>
<td>15 0 0 14</td>
</tr>
<tr>
<td>DPI</td>
<td>14 0</td>
<td>4 0 0 0</td>
<td>0 0 0 1</td>
</tr>
<tr>
<td>CDPR</td>
<td>13 0</td>
<td>13 0 0</td>
<td>15 0 0 14</td>
</tr>
</tbody>
</table>

DPI/DPR = Data Proc. immd./reg 2nd operand
CDPR = Cond. DPR
BR = Branch

DPI 14: cmp r2, #0
DPI 14: mov r0, #0
DPI 14: movle r15, r14
DPI 14: subs r2, r2, #1
DPI 14: add r0, r0, r1
DPI 14: movle r15, r14
DPI 14: b 4194316 ;(pc-4*5)

### Decoding Example (#6/7)

<table>
<thead>
<tr>
<th>Binary</th>
<th>Fields</th>
<th>Decimal</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPI</td>
<td>14 1</td>
<td>10 1 2</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>DPI</td>
<td>14 1</td>
<td>13 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>CDPR</td>
<td>13 0</td>
<td>13 0 0</td>
<td>15 0 0 14</td>
</tr>
<tr>
<td>DPI</td>
<td>14 0</td>
<td>4 0 0 0</td>
<td>0 0 0 1</td>
</tr>
<tr>
<td>CDPR</td>
<td>13 0</td>
<td>13 0 0</td>
<td>15 0 0 14</td>
</tr>
</tbody>
</table>

DPI/DPR = Data Proc. immd./reg 2nd operand
CDPR = Cond. DPR
BR = Branch

DPI 14: cmp r2, #0
DPI 14: mov r0, #0
DPI 14: movle r15, r14
DPI 14: subs r2, r2, #1
DPI 14: add r0, r0, r1
DPI 14: movle r15, r14
DPI 14: b 4194316 ;Loop@pc-20
DPI 14: subs r2, r2, #1
DPI 14: add r0, r0, r1
DPI 14: movle r15, r14
DPI 14: b 4194332:
DPI 14: subs r2, r2, #1
DPI 14: add r0, r0, r1
DPI 14: movle r15, r14
DPI 14: b Loop

### Decoding Example (#7/7)

<table>
<thead>
<tr>
<th>Binary</th>
<th>Fields</th>
<th>Decimal</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPI</td>
<td>14 1</td>
<td>10 1 2</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>DPI</td>
<td>14 1</td>
<td>13 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>CDPR</td>
<td>13 0</td>
<td>13 0 0</td>
<td>15 0 0 14</td>
</tr>
<tr>
<td>DPI</td>
<td>14 0</td>
<td>4 0 0 0</td>
<td>0 0 0 1</td>
</tr>
<tr>
<td>CDPR</td>
<td>13 0</td>
<td>13 0 0</td>
<td>15 0 0 14</td>
</tr>
</tbody>
</table>

DPI/DPR = Data Proc. immd./reg 2nd operand
CDPR = Cond. DPR
BR = Branch

DPI 14: cmp a3, #0
DPI 14: mov a1, #0
DPI 14: movle pc, lr
DPI 14: subs a3, a3, #1
DPI 14: add a1, a1, a2
DPI 14: movle pc, lr
DPI 14: b Loop

### Instruction Set Bridge

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

![Diagram showing ARM Machine Language and its interactions with other languages and tools](image)

- GCC
- Fortran
- Java
- ARM Machine Language
- ARM-elf/ARM-AXD
- ARM Assembly Language
- ARM-elf
- PC GDB-Debug interpreter
- 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