ELEC2041

Microprocessors and Interfacing

Lectures 23: Instruction Representation; Assembly and Decoding

http://webct.edtec.unsw.edu.au/

May 2006

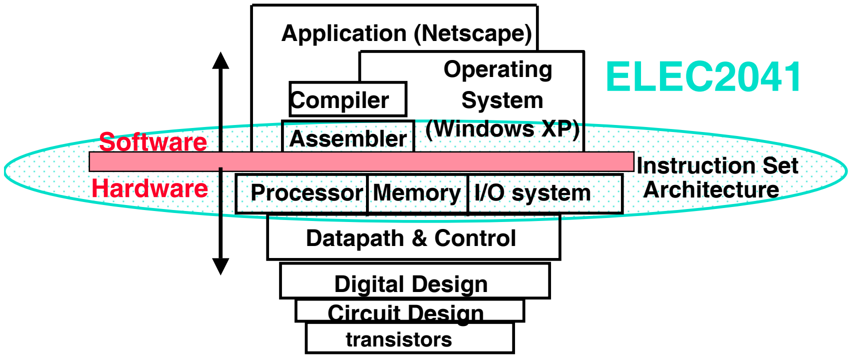
Saeid Nooshabadi

saeid@unsw.edu.au

Overview

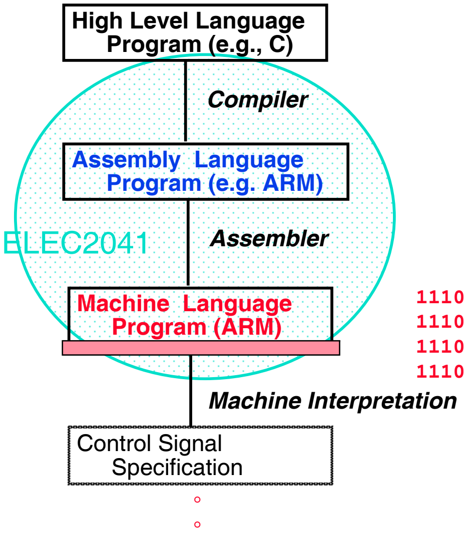
- What computers really do
  - fetch / decode / execute cycle
- Assembly: action ➔ to bits
- Decoding: bits ➔ actions
- Disassembly
- Conclusion

Review: What is Subject about?



- Coordination of many *levels of abstraction*

Review:Programming Levels of Representation



```
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 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
```

ALUOP[0:3] <= InstReg[9:11] & MASK

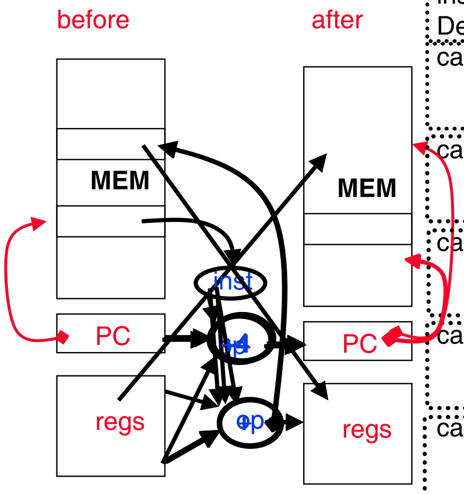
Review: What Does a Computer Do?

- 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

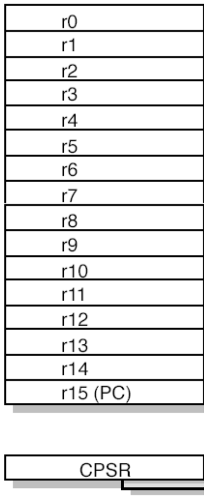
Fetch/Decode/Execute Cycle

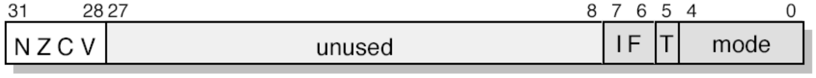


```
inst ← Fetch MEM[ PC ]
Decode( inst )
case ARITH/LOG
        $REG_{inst}$ ← $REG_{inst}$ $OP_{inst}$ $REG_{inst}$
        PC ← PC + 4
case ARITH/LOG-immed
        $REG_{inst}$ ← $REG_{inst}$ $OP_{inst}$ $IM_{inst}$
        PC ← PC + 4
case LOAD
        $REG_{inst}$ ← MEM[ $REG_{inst}$ + $IM_{inst}$ ]
        PC ← PC + 4
case STORE
        MEM[ $REG_{inst}$ + $IM_{inst}$ ] ← $REG_{inst}$
        PC ← PC + 4
case CONTROL
        PC ← $OP_{inst}$(PC, $REG_{inst}$, $IM_{inst}$ )
```

Review: Instruction Set (ARM 7TDMI)

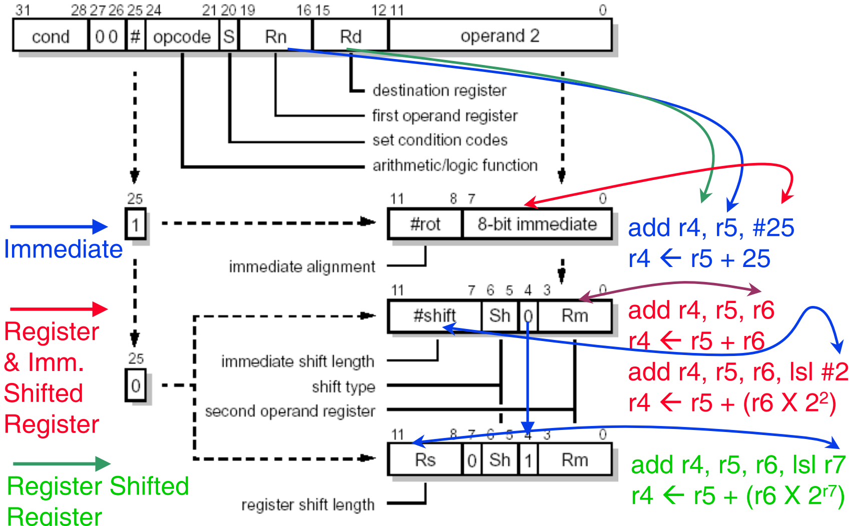
- Set of instruction that a processor can execute
- Instruction Categories
  - Data Processing or Computational (Logical and Arithmetic
  - Load/Store (Memory Access)
  - Control Flow (Jump and Branch)
  - Floating Point
    - coprocessor
  - Memory Management
  - Special





ARM Data Processing Instructions

All instructions 32 bits wide



Immediate

add r4, r5, #25
r4 ← r5 + 25

Register & Imm. Shifted Register

add r4, r5, r6
r4 ← r5 + r6
add r4, r5, r6, lsl #2
r4 ← r5 + (r6 X 2^2)

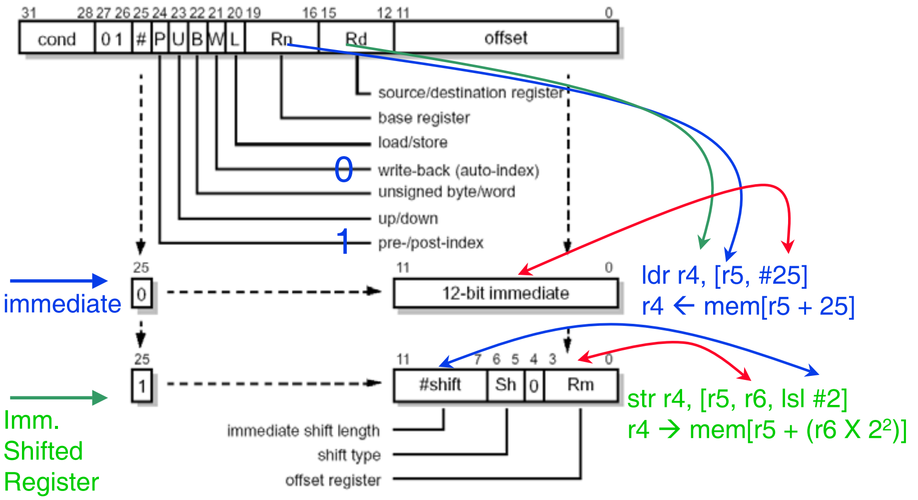
Register Shifted Register

add r4, r5, r6, lsl r7
r4 ← r5 + (r6 X 2^{r7})

3 types of addressing modes

ARM Load/Store Instructions (#1/3)

All instructions 32 bits wide



immediate

ldr r4, [r5, #25]
r4 ← mem[r5 + 25]

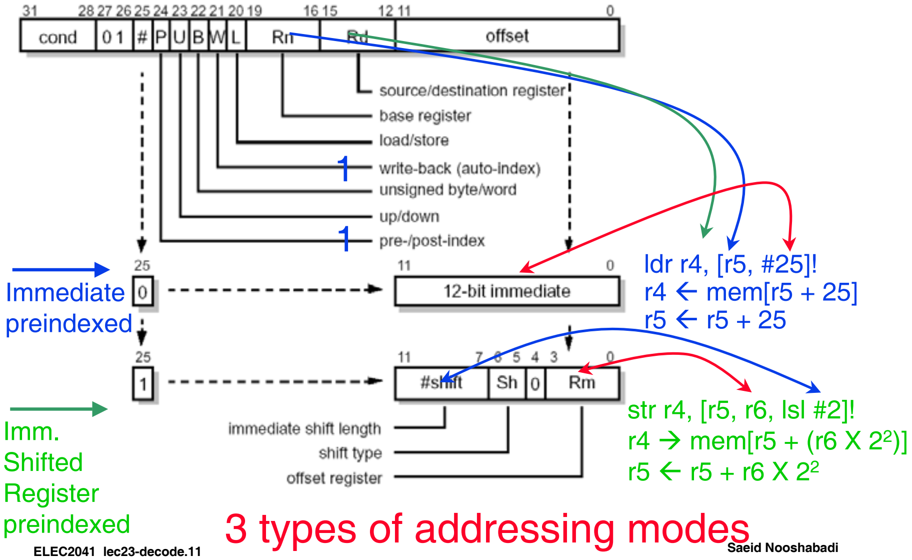
Imm. Shifted Register

str r4, [r5, r6, lsl #2]
r4 → mem[r5 + (r6 X 2^2)]

3 types of addressing modes

ARM Load/Store Instructions (#2/3)

All instructions 32 bits wide



Immediate preindexed

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

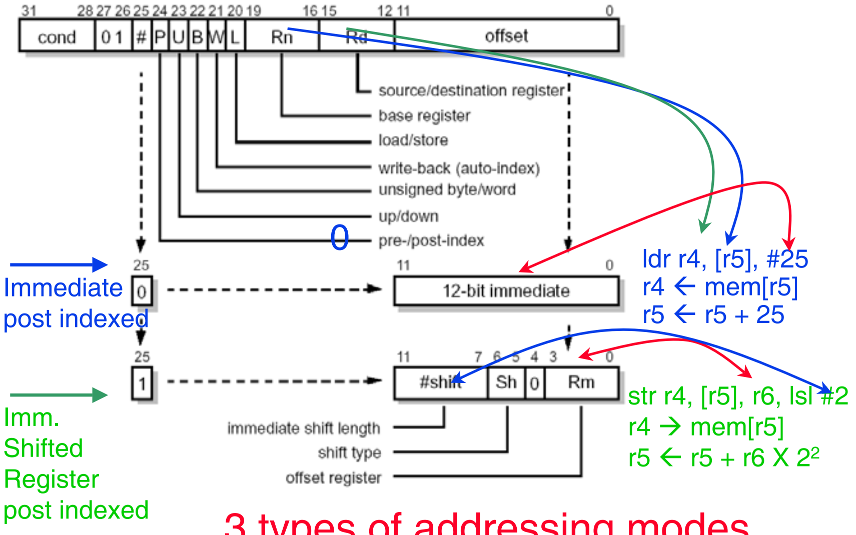
Imm. Shifted Register preindexed

str r4, [r5, r6, lsl #2]!
r4 → mem[r5 + (r6 X 2^2)]
r5 ← r5 + r6 X 2^2

3 types of addressing modes

ARM Load/Store Instructions (#3/3)

All instructions 32 bits wide



Immediate post indexed

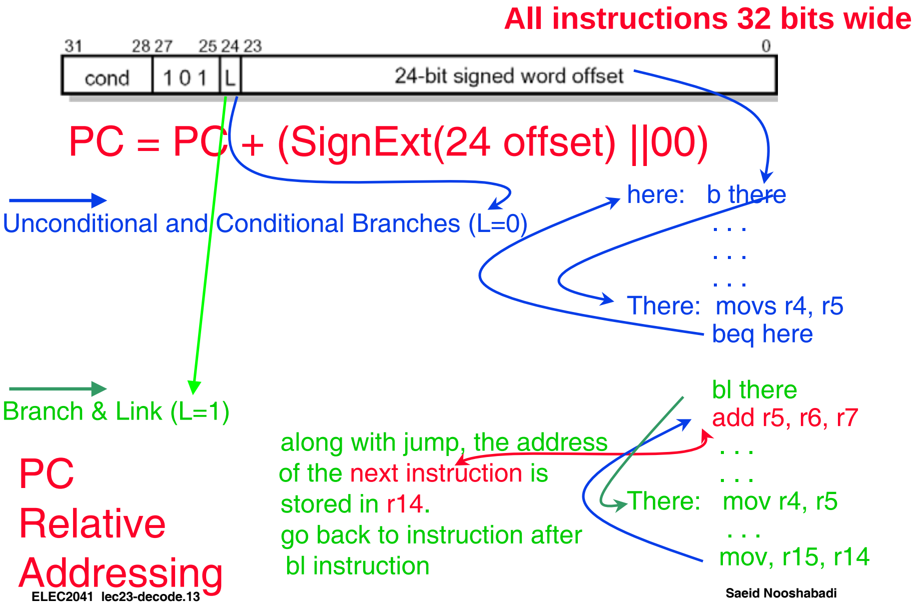
ldr r4, [r5], #25
r4 ← mem[r5]
r5 ← r5 + 25

Imm. Shifted Register post indexed

str r4, [r5], r6, lsl #2
r4 → mem[r5]
r5 ← r5 + r6 X 2^2

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
        beq here
```

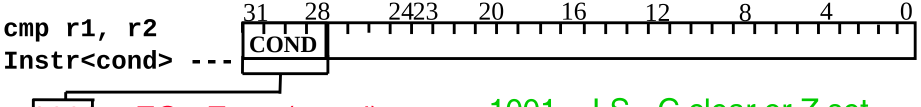
Branch & Link (L=1)

PC Relative Addressing

along with jump, the address of the next instruction is stored in r14.
go back to instruction after bl instruction

```
        bl there
        add r5, r6, r7
        . . .
        . . .
There:  mov r4, r5
        . . .
        mov, r15, r14
```

Conditional Execution Field



```
cmp r1, r2
Instr<cond> ---
```

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 >or =)

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.

ARM Instruction Set Format

31	2827	1615		87		0	**Instruction type**
Cond	0 0 I Opcode S	Rn	Rd	Operand2			Data processing / PSR transfer
Cond	0 0 0 0 0 0 A S	Rd	Rn	Rs	1 0 0 1	Rm	Multiply
Cond	0 0 0 0 1 U A S	RdHi	RdLo	Rs	1 0 0 1	Rm	Long Multiply **(v3M / v4 only)**
Cond	0 0 0 1 0 B 0 0	Rn	Rd	0 0 0 0	1 0 0 1	Rm	Swap
Cond	0 1 I P U B W L	Rn	Rd	Offset			Load/Store Byte/Word
Cond	1 0 0 P U S W L	Rn	Register List				Load/Store Multiple
Cond	0 0 0 P U 1 W L	Rn	Rd	Offset1	1 S H 1	Offset2	Halfword transfer:Immediate offset **(v4 only)**
Cond	0 0 0 P U 0 W L	Rn	Rd	0 0 0 0	1 S H 1	Rm	Halfword transfer: Register offset **(v4 only)**
Cond	1 0 1 L	Offset					Branch
Cond	0 0 0 1 0 0 1 0	1 1 1 1	1 1 1 1	1 1 1 1	0 0 0 1	Rn	Branch Exchange **(v4T only)**
Cond	1 1 0 P U N W L	Rn	CRd	CPNum	Offset		Coprocessor data transfer
Cond	1 1 1 0 Op1	CRn	CRd	CPNum	Op2 0	CRm	Coprocessor data operation
Cond	1 1 1 0 Op1 L	CRn	Rd	CPNum	Op2 1	CRm	Coprocessor register transfer
Cond	1 1 1 1	SWI Number					Software interrupt

Reading Material

- **Steve Furber: ARM System On-Chip; 2nd Ed, Addison-Wesley, 2000, ISBN: 0-201-67519-6. chapter 5**
- **ARM Architecture Reference Manual 2nd Ed, Addison-Wesley, 2001, ISBN: 0-201-73719-1,, chapter A2: Programmer's Model**

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:

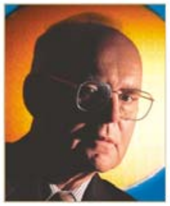
1. Mother of A laws: Moore’s Law

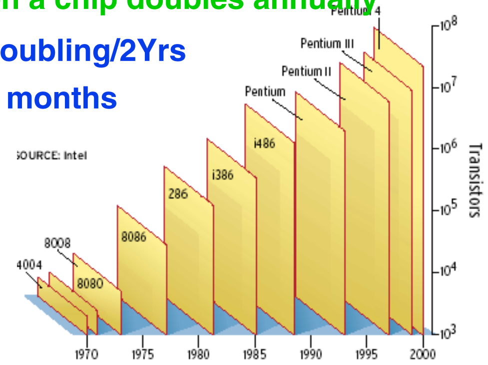
Suggested by Intel Corp. legend Gordon E. Moore 38 years ago.

The number of transistors on a chip doubles annually

The current growth rate is doubling/2Yrs

Intel PR quotes doubling/18 months





---

5 Rules that Comp Engineers Live by (#2/5)

2. Rock’s Law

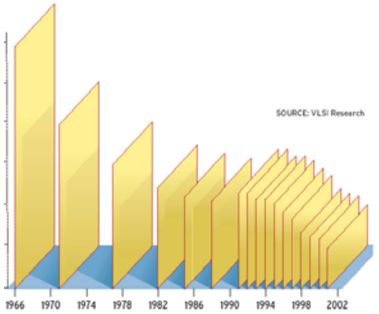
Suggested by Intel Corp. investor Arthur Rock

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),



---

5 Rules that Comp Engineers Live by (#3/5)

3. MACHRONE’S Law

Suggested by 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.”

1984: IBM 5155 Intel 8088 at 4.77 MHz, $4225, amber monochrome monitor and no hard drive

2003: Apple Macintosh G5, 2 GHz, about $5000, with largest flat-panel display and fully loaded

---

5 Rules that Comp Engineers Live by (#4/5)

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 NiklausWirth of ETH Switzerland), inventor of the Pascal computer language,

Software is slowing faster than hardware is accelerating

"Groves giveth, and Gates take 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: ℘〉ΣΣ| .

Users tolerate "feature bloat" for reasons:

1. Moore's Law, which makes the bloat possible,

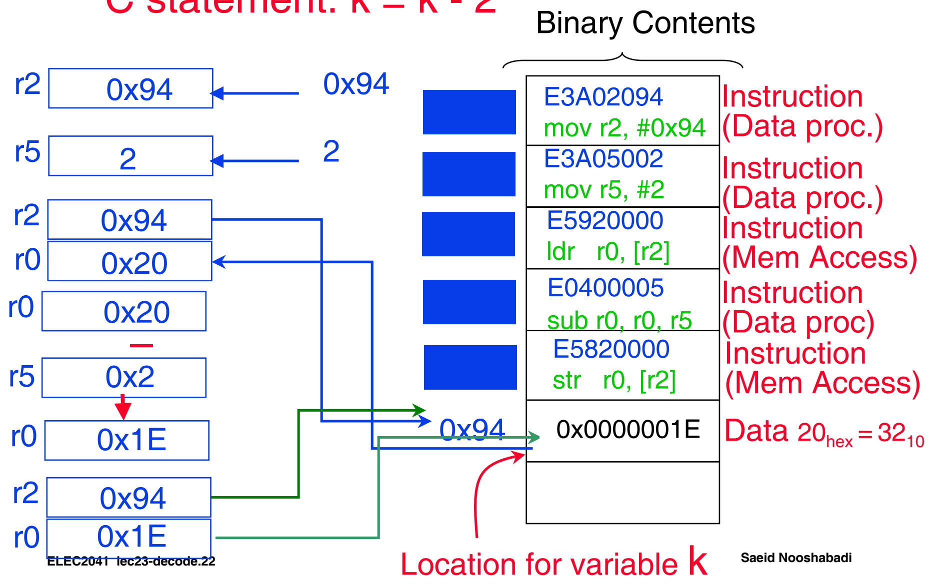
2. Ignorance among consumers

Source: IEEE Spectrum Dec 2003

The root cause is the interests of software companies

Recall: Sample Assembly Program

C statement: k = k - 2



Binary Contents

E3A02094 mov r2, #0x94	Instruction (Data proc.)
E3A05002 mov r5, #2	Instruction (Data proc.)
E5920000 ldr r0, [r2]	Instruction (Mem Access)
E0400005 sub r0, r0, r5	Instruction (Data proc)
E5820000 str r0, [r2]	Instruction (Mem Access)
0x0000001E	Data $20_{hex} = 32_{10}$

Location for variable k

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, 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.**

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$_{10}$ (0x00400000).

- Next step: convert to binary

Decoding Example (#2/7)

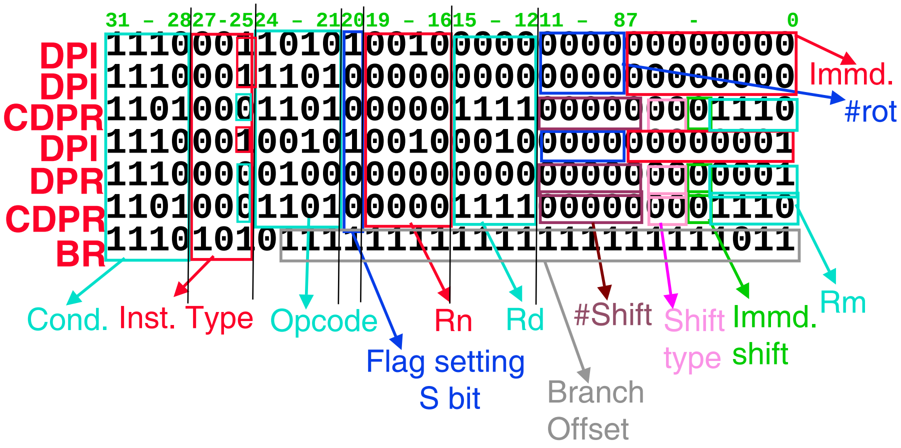
- **Binary** ⇒ Decimal ⇒ Assembly ⇒ C?
- Start at program at address 4,194,304$_{10}$ = 0x00400000 (2^{22})

```
11100011010100100000000000000000
11100011101000000000000000000000
11010001101000001111000000001110
11100010010100100010000000000001
11100000100000000000000000000001
11010001101000001111000000001110
11101010111111111111111111111011
```

- What are instruction formats of these 7 instructions?

Decoding Example (#3/7)

Binary ⇒ **Fields** ⇒ Decimal ⇒ Assembly ⇒ C?



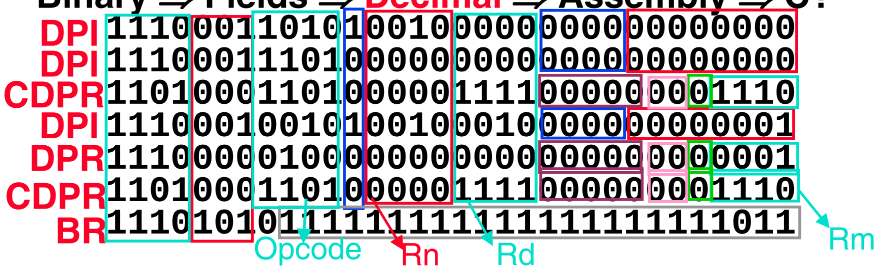
DPI/DPR = Data Proc. immd./reg 2nd opernd

CDPR = Cond. DPR

BR = Branch

Decoding Example (#4/7)

Binary ⇒ Fields ⇒ **Decimal** ⇒ Assembly ⇒ C?



DPI/DPR = Data Proc. immd./reg 2nd opernd

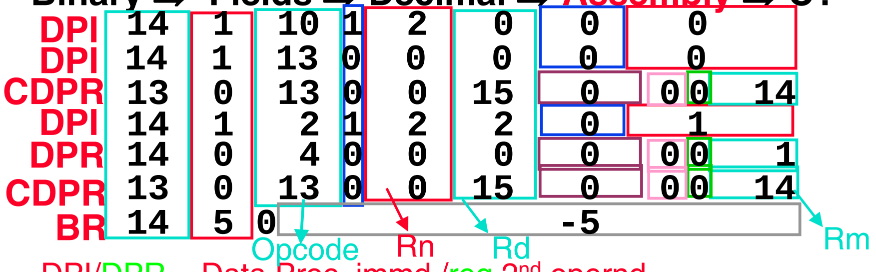
CDPR = Cond. DPR

BR = Branch

DPI	14	1	10	1	2	0	0			0
DPI	14	1	13	0	0	0	0			0
CDPR	13	0	13	0	0	15	0	0	0	14
DPI	14	1	2	1	2	2	0			1
DPR	14	0	4	0	0	0	0	0	0	1
CDPR	13	0	13	0	0	15	0	0	0	14
BR	14	5	0	-5						

Decoding Example (#5/7)

Binary ⇒ Fields ⇒ Decimal ⇒ Assembly ⇒ C?



DPI	14	1	10	1	2	0	0	0		
DPI	14	1	13	0	0	0	0	0		
CDPR	13	0	13	0	0	15	0	0	0	14
DPI	14	1	2	1	2	2	0	1		
DPR	14	0	4	0	0	0	0	0	0	1
CDPR	13	0	13	0	0	15	0	0	0	14
BR	14	5	0	-5						

DPI/DPR = Data Proc. immd./reg 2nd opernd

CDPR = Cond. DPR BR = Branch

```
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
4194328: b        4194316 ;(pc-4*5)
```

Decoding Example (#6/7)

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

recall: branch target computed by adding offset (<<2) to address of branch inst plus 8

```
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
4194328: b        4194316 ;Loop@pc-20
4194332:
4194336:                  ;pc=419336-20 =4194316
```

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

Decoding Example (#7/7)

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

A
R
M

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

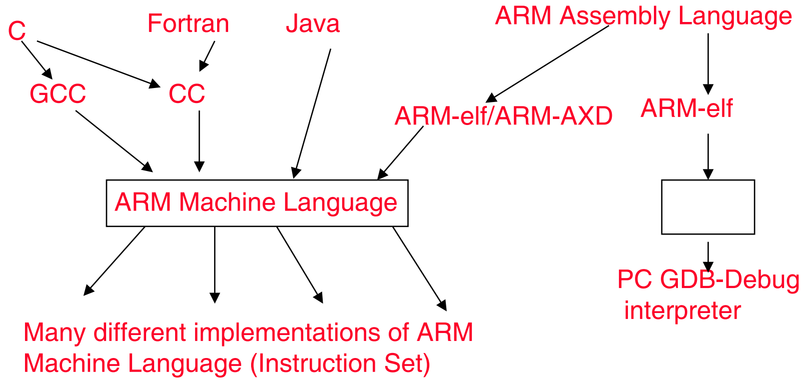
Mapping `product:a1, mcand:a2, mlier:a3;`

C

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

Instruction Set Bridge

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



"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