EEEN301 Embedded systems

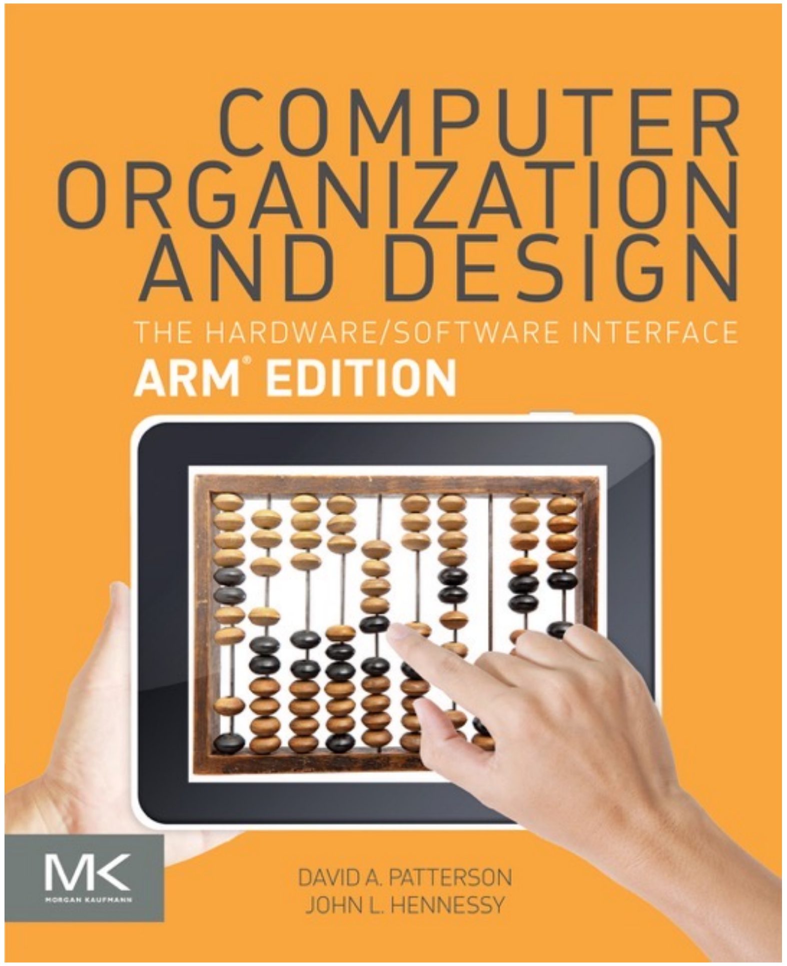
Lecture 3 2023

Computer organization

Language of the computer

Reference book

David A. Patterson and John L. Hennessy, Computer Organization and Design, ARM edition, Morgan Kaufmann Publishers.



The Cortex-A9 processor used in the Beaglebone implements the ARMv7-A architecture and instruction set.

Newer and larger ‘Arm Cortex’ processors now use the ARMv8-A architecture and instruction set. The set text book also covers the ARMv8-A so we will examine that instruction set.

The new Apple ’M1” processor is a 64bit ARM processor that uses the ARMv8.4-A instruction set.

Chapter 2

Instructions: Language of the Computer

Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

The ARMv8 Instruction Set

- A subset, called LEGv8, used as the example throughout the book
- Commercialized by ARM Holdings (www.arm.com)
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern ISAs
  - See ARM Reference Data tear-out card

Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination

```
ADD a, b, c  // a gets b + c
```

- All arithmetic operations have this form
- *Design Principle 1:* Simplicity favours regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Register Operands

- Arithmetic instructions use register operands
- LEGv8 has a 32 × 64-bit register file
  - Use for frequently accessed data
  - 64-bit data is called a “doubleword”
    - 31 x 64-bit general purpose registers X0 to X30
  - 32-bit data called a “word”
    - 31 x 32-bit general purpose sub-registers W0 to W30
- *Design Principle 2:* Smaller is faster
  - c.f. main memory: millions of locations

LEGv8 Registers

- X0 – X7: procedure arguments/results
- X8: indirect result location register
- X9 – X15: temporaries
- X16 – X17 (IP0 – IP1): may be used by linker as a scratch register, other times as temporary register
- X18: platform register for platform independent code; otherwise a temporary register
- X19 – X27: saved
- X28 (SP): stack pointer
- X29 (FP): frame pointer
- X30 (LR): link register (return address)
- XZR (register 31): the constant value 0

Register Operand Example

- C code:

```
f = (g + h) - (i + j);
```

  - f, …, j in X19, X20, …, X23

- Compiled LEGv8 code:

```
ADD X9, X20, X21
ADD X10, X22, X23
SUB X19, X9, X10
```

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- LEGv8 does not require words to be aligned in memory, except for instructions and the stack

Memory Operand Example

- C code:

```
A[12] = h + A[8];
```

  - h in X21, base address of A in X22

- Compiled LEGv8 code:
  - Index 8 requires offset of 64

```
LDUR    X9,[X22,#64] // U for "unscaled"
ADD     X9,X21,X9
STUR    X9,[X22,#96]
```

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

Immediate Operands

- Constant data specified in an instruction

```
ADDI X22, X22, #4
```

- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: 0 to $+2^n - 1$
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
  - 0 to +4,294,967,295

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^{2} + 0\times2^{1} + 0\times2^{0}$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits
  - –2,147,483,648 to +2,147,483,647

2s-Complement Signed Integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- $-(-2^{n-1})$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

Signed Negation

- Complement and add 1
  - Complement means 1 → 0, 0 → 1

$$x + \bar{x} = 1111...111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Example: negate +2
  - $+2 = 0000\ 0000\ \ldots\ 0010_{two}$
  - $-2 = 1111\ 1111\ \ldots\ 1101_{two} + 1$
    $= 1111\ 1111\ \ldots\ 1110_{two}$

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110
- In LEGv8 instruction set
  - `LDURSB`: sign-extend loaded byte
  - `LDURB`: zero-extend loaded byte

Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- LEGv8 instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!

Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

LEGv8 R-format Instructions

opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

- Instruction fields
  - opcode: operation code
  - Rm: the second register source operand
  - shamt: shift amount (00000 for now)
  - Rn: the first register source operand
  - Rd: the register destination

R-format Example

opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

ADD `X9,X20,X21`

1112_{ten}	21_{ten}	0_{ten}	20_{ten}	9_{ten}
10001011000_{two}	10101_{two}	000000_{two}	10100_{two}	01001_{two}

$1000\ 1011\ 0001\ 0101\ 0000\ 0010\ 1000\ 1001_{two}$ =

$8B150289_{16}$

LEGv8 D-format Instructions

opcode	address	op2	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits

- Load/store instructions
  - Rn: base register
  - address: constant offset from contents of base register (+/- 32 doublewords)
  - Rt: destination (load) or source (store) register number
- *Design Principle 3:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

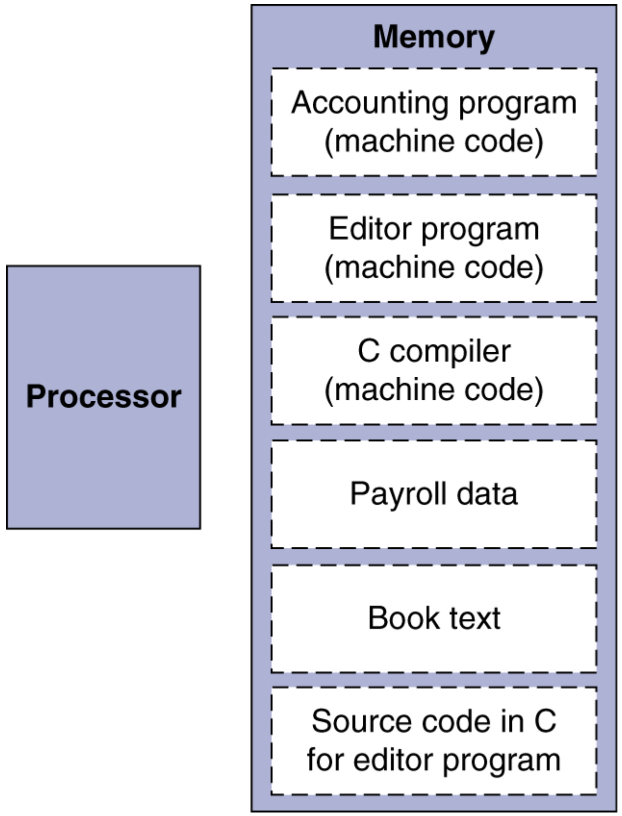
LEGv8 I-format Instructions

opcode	immediate	Rn	Rd
10 bits	12 bits	5 bits	5 bits

- Immediate instructions
  - Rn: source register
  - Rd: destination register
- Immediate field is zero-extended

Stored Program Computers

The BIG Picture


Memory
Accounting program (machine code)
Editor program (machine code)
C compiler (machine code)
Payroll data
Book text
Source code in C for editor program
Processor

- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	LEGv8
Shift left	<<	<<	`LSL`
Shift right	>>	>>>	`LSR`
Bit-by-bit AND	&	&	`AND, ANDI`
Bit-by-bit OR	|	|	`OR, ORI`
Bit-by-bit NOT	~	~	`EOR, EORI`

- Useful for extracting and inserting groups of bits in a word

Shift Operations

opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

- shamt: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - `LSL` by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - `LSR` by *i* bits divides by 2^i (unsigned only)

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

```
AND X9,X10,X11
```

X10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
X11	00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
X9	00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000

OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```
OR X9,X10,X11
```

X10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
X11	00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
X9	00000000 00000000 00000000 00000000 00000000 00000000 00111101 11000000

EOR Operations

- Differencing operation
  - Set some bits to 1, leave others unchanged

```
EOR X9,X10,X12  // NOT operation
```

X10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
X12	11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111
X9	11111111 11111111 11111111 11111111 11111111 11111111 11110010 00111111

Conditional Operations

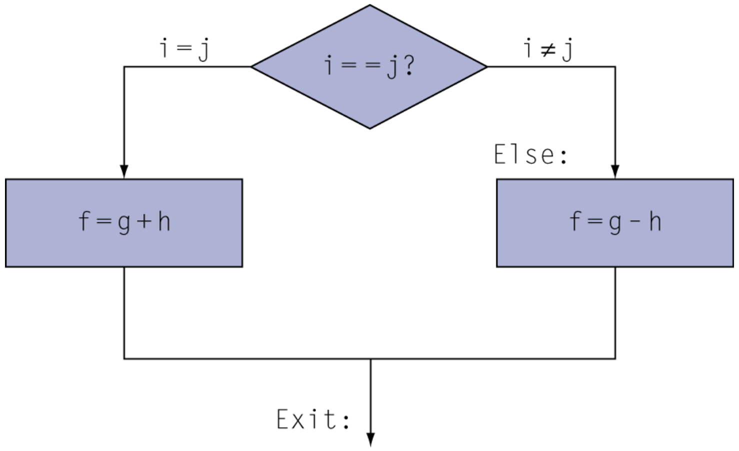
- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `CBZ register, L1`
  - if (register == 0) branch to instruction labeled L1;
- `CBNZ register, L1`
  - if (register != 0) branch to instruction labeled L1;
- `B L1`
  - branch unconditionally to instruction labeled L1;

Compiling If Statements

- C code:

```
if (i==j) f = g+h;
else f = g-h;
```

  - f, g, … in X19, X20, …



- Compiled LEGv8 code:

```
      SUB X9,X22,X23
      CBNZ X9,Else
      ADD X19,X20,X21
      B Exit
Else: SUB X19,X20,x21
Exit: ...
```

Assembler calculates addresses

Compiling Loop Statements

- C code:

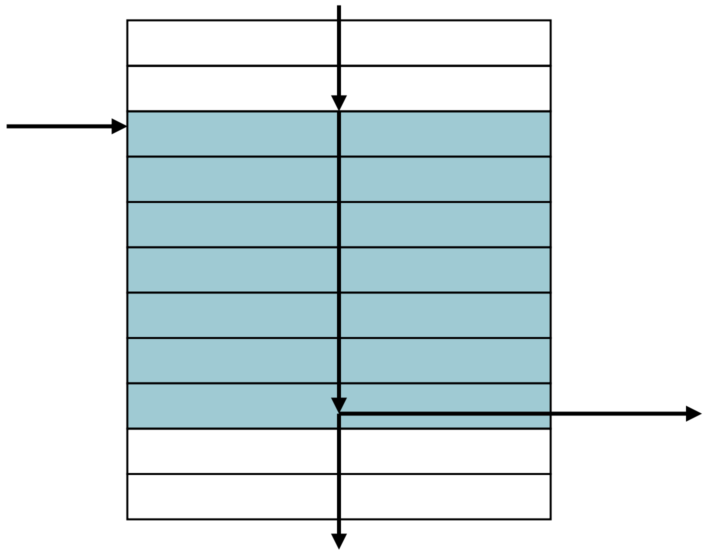
```
while (save[i] == k) i += 1;
```

  - i in x22, k in x24, address of save in x25
- Compiled LEGv8 code:

```
Loop: LSL    X10,X22,#3
      ADD    X10,X10,X25
      LDUR   X9,[X10,#0]
      SUB    X11,X9,X24
      CBNZ   X11,Exit
      ADDI   X22,X22,#1
      B      Loop
Exit: …
```

Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

More Conditional Operations

- Condition codes, set from arithmetic instruction with S-suffix (ADDS, ADDIS, ANDS, ANDIS, SUBS, SUBIS)
  - negative (N): result had 1 in MSB
  - zero (Z): result was 0
  - overlow (V): result overflowed
  - carry (C): result had carryout from MSB
- Use subtract to set flags, then conditionally branch:
  - **B.EQ**
  - **B.NE**
  - **B.LT** (less than, signed), **B.LO** (less than, unsigned)
  - **B.LE** (less than or equal, signed), **B.LS** (less than or equal, unsigned)
  - **B.GT** (greater than, signed), **B.HI** (greater than, unsigned)
  - **B.GE** (greater than or equal, signed),
  - **B.HS** (greater than or equal, unsigned)

Conditional Example

- if (a > b) a += 1;
  - a in X22, b in X23

```
      SUBS X9,X22,X23  // use subtract to make comparison
      B.LTE Exit       // conditional branch
      ADDI X22,X22,#1
Exit:
```

Signed vs. Unsigned

- Signed comparison
- Unsigned comparison
- Example
  - X22 = 1111 1111 1111 1111 1111 1111 1111 1111
  - X23 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `x22 < x23 # signed`
    - –1 < +1
  - `x22 > x23 # unsigned`
    - +4,294,967,295 > +1