Chapter 2

Instructions: Language of the Computer

Tutorial 2

- Unsigned Binary Integers and 2s-Complement
- Signed Negation and Signed Extension
- Stored Program Computers
- Representing Instructions
- Logical Operations
- Conditional Operations
  - Compiling If Statements
  - Compiling Loop Statements

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

$$x + \overline{x} = 1111\ldots111_2 = -1$$

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

- Example: negate +2
  - $+2 = 0000\ 0000\ \ldots\ 0010_2$
  - $-2 = 1111\ 1111\ \ldots\ 1101_2 + 1$
    $= 1111\ 1111\ \ldots\ 1110_2$

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

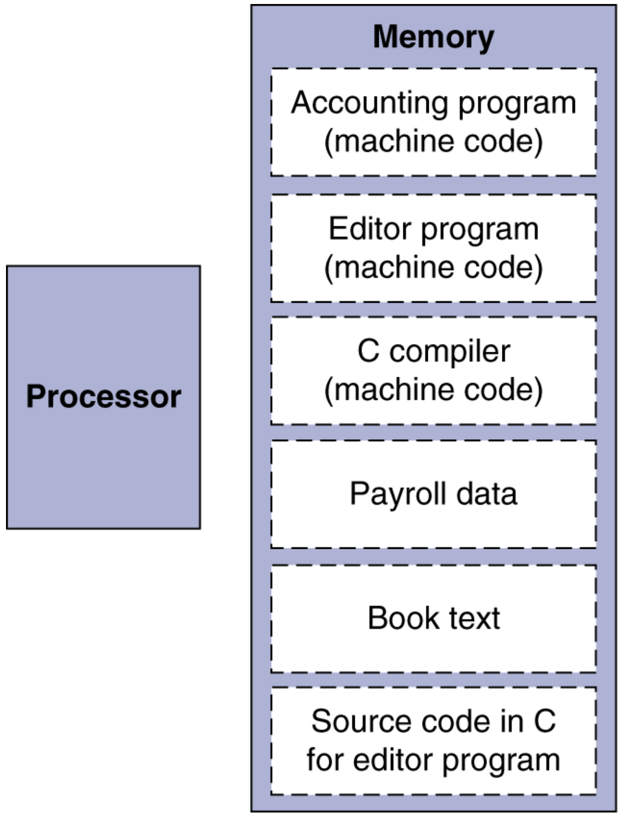
Sign Extension in ARM

- In ARM instruction set

Instruction	Example	Meaning
load register halfword signed	LDRHS r1, [r2,#20]	r1= Memory[r2 + 20]
load register byte signed	LDRBS r1, [r2,#20]	r1= Memory[r2 + 20]

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

Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- ARM instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!
- Register numbers – r0 to r15

ARM Data Processing (DP) Instructions

Cond	F	I	Opcode	S	Rn	Rd	Operand2
4 bits	2 bits	1 bits	4 bits	1 bits	4 bits	4 bits	12 bits

- Instruction fields
  - Opcode : Basic operation of the instruction
  - Rd: The destination register operand
  - Rn: The first register source operand
  - Operand2: The second source operand
  - I:Immediate. If I is 0, the second source operand is a register, else the second source is a 12-bit immediate.
  - S: Set Condition Code
  - Cond: Condition
  - F: Instruction Format.

DP Instruction Example

Cond	F	I	Opcode	S	Rn	Rd	Operand2
4 bits	2 bits	1 bits	4 bits	1 bits	4 bits	4 bits	12 bits

```
ADD r5,r1,r2 ; r5 = r1 + r2
```

14	0	0	4	0	1	5	2
4 bits	2 bits	1 bits	4 bits	1 bits	4 bits	4 bits	12 bits

$11100000100000010101000000000010_2$

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

ARM Data Transfer (DT) Instruction

Cond	F	Opcode	Rn	Rd	Offset12
4 bits	2 bits	6 bits	4 bits	4 bits	12 bits

LDR r5, [r3, #32]   ; Temporary reg r5 gets A[8]

14	1	24	3	5	32
4 bits	2 bits	6 bits	4 bits	4 bits	12 bits

Design Principle 4

- Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	ARM
Bitwise AND	&	&	AND
Bitwise OR	|	|	ORR
Bitwise NOT	~	~	MVN
Shift left	<<	<<	LSL
Shift right	>>	>>	LSR

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

AND Operations

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

```
AND r5, r1, r2 ; reg r5 = reg r1 & reg r2
```

```
r2    0000 0000 0000 0000 0000 1101 1100 0000
r1    0000 0000 0000 0000 0011 1100 0000 0000
r5    0000 0000 0000 0000 0000 1100 0000 0000
```

OR Operations

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

```
ORR r5, r1, r2  ; reg r5 = reg r1 | reg r2
```

r2	0000 0000 0000 0000 0000 1101 1100 0000
r1	0000 0000 0000 0000 0011 1100 0000 0000
r5	0000 0000 0000 0000 0011 1101 1100 0000

NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- ARM has Move Not (MVN)

```
MVN r5, r1 ;  reg r5 = ~ reg r1
```

r1	0000 0000 0000 0000 0011 1100 0000 0000
r5	1111 1111 1111 1111 1100 0011 1111 1111

Shift Operations

- Logical shift left (LSL)
  - Shift left and fill with 0 bits
  - LSL by *i* bits multiplies by 2^i
- Logical shift right(LSR)
  - Shift right and fill with 0 bits
  - LSR by *i* bits divides by 2^i (unsigned only)

Shift Operations

- Although ARM has shift operations, they are not separate instructions
- ARM offers the **ability** to shift the second operand as part of any data processing instruction!
- Examples
  - *ADD r5,r1,r2, LSL #2 ; r5 = r1 + (r2 << 2)*
  - *MOV r6,r5, LSR #4 ; r6 = r5 >> 4*
  - *MOV r6,r5, LSR r3 ; r6 = r5 >> r3*

Shift Operations

- Still the assembler supports instructions such as
  - LSR *r6,r5*,#4 *; r6 = r5 >> 4*
- They are synonyms for MOV instructions with shifted register second operands
  - *MOV r6,r5, LSR #4 ; r6 = r5 >> 4*

Shift Operations

Cond	000	Opcode	S	Rn	Rd	shift_imm	ST	Rm
4 bits	3 bits	4 bits	1 bit	4 bits	4 bits	5 bits	3 bits	4 bits

- shift_imm: how many positions to shift
- Logical shift left (LSL)
  - *<OP> Rd, Rn, Rm, LSL #<shift_imm>*
  - *Eg : ADD r5, r1, r2, LSL #2 ; r5 = r1 + (r2 << 2)*
- Logical shift right(LSR)
  - *<OP> Rd, Rn, Rm, LSR #<shift_imm>*
  - *Eg : ADD r5, r1, r2, LSR #2 ; r5 = r1 + (r2 << 2)*

Conditional Operations

- What distinguishes a computer from a simple calculator is its ability to make decisions.
- Based on the input data and the values created during computation, different instructions execute.
    - Eg : if conditions, loops
- Branch to a labeled instruction if a condition is true
    - Otherwise, continue sequentially

Conditional Branches

```
CMP reg1,reg2
BEQ L1
```

- Mnemonic CMP stands for compare. It sets the Program Status Register
- BEQ stands for Branch if Equal
- if (reg1 == reg2) branch to instruction labeled L1;

```
CMP reg1,reg2
BNE L1
```

- BNE stands for Branch if Not Equal
- if (reg1 != reg2) branch to instruction labeled L1;

Unconditional Branch

- `B exit      ; go to exit`
  - unconditional jump to instruction labeled exit to instruction labeled L1;

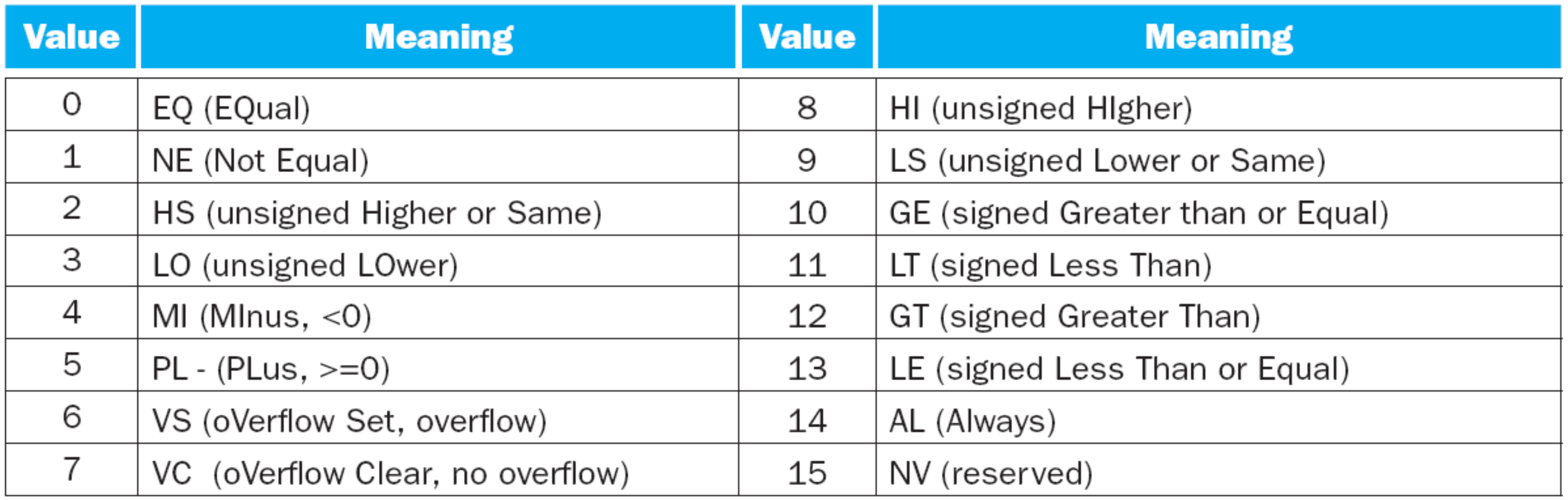
Signed vs. Unsigned

- Signed comparison: `BGE,BLT,BGT,BLE`
- Unsigned comparison: `BHS,BLO,BHI,BLS`
- Example
  - r0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - r1 = 0000 0000 0000 0000 0000 0000 0000 0001
    - `CMP r0,r1`
    - `BLO L1 ; unsigned branch`
      - Branch not taken since $4{,}294{,}967{,}295_{ten} > 1_{ten}$
    - `BLT L2  ; signed branch`
      - Branch taken to L2 since $-1_{ten} < 1_{ten}$.

Branch Instruction format

Condition	12	address
4 bits	4 bits	24 bits

- Encoding of options for Condition field



Value	Meaning	Value	Meaning
0	EQ (EQual)	8	HI (unsigned HIgher)
1	NE (Not Equal)	9	LS (unsigned Lower or Same)
2	HS (unsigned Higher or Same)	10	GE (signed Greater than or Equal)
3	LO (unsigned LOwer)	11	LT (signed Less Than)
4	MI (MInus, <0)	12	GT (signed Greater Than)
5	PL - (PLus, >=0)	13	LE (signed Less Than or Equal)
6	VS (oVerflow Set, overflow)	14	AL (Always)
7	VC (oVerflow Clear, no overflow)	15	NV (reserved)

PC-relative addressing

- 24-bit address
- Problem :
  - 2^{24} or 16 MB is not enough for large programs
- Solution :
  - Program counter (PC) = Register + Branch address
  - More than 2^{32} memory locations
  - But which Register?

PC-relative addressing

- Conditional branches in loops and if statements, branch to a nearby instruction.
  - Generally less than 16 instructions away
- Hence PC= PC ± Branch address
- All ARM instructions are 4 bytes
  - Refer to the number of *words* to the next instruction instead of the number of bytes
  - Relative distance increase by 4-times

Conditional Execution

- Before

```
      CMP r3,r4
      BNE Else               ; go to Else if i ≠ j
      ADD r0,r1,r2           ; f = g + h (skipped if i ≠ j)
      B Exit                 ; go to Exit
Else:
      SUB r0,r1,r2           ; f = g - h (skipped if i = j )
Exit:
```

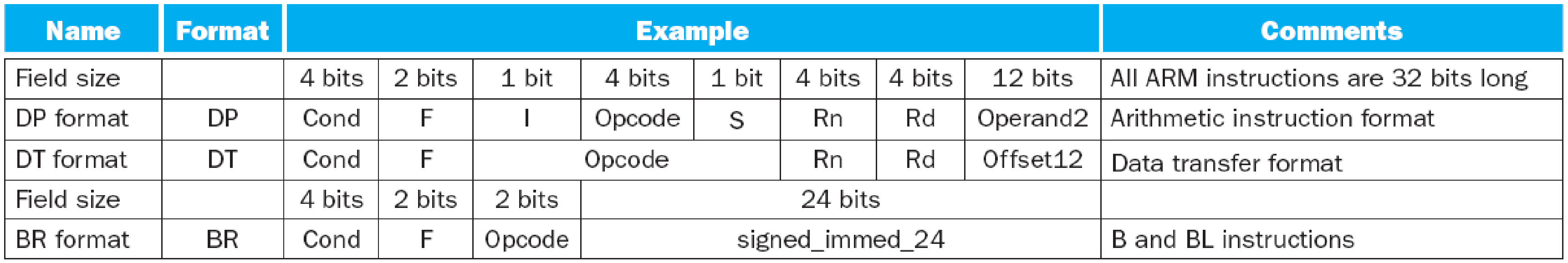
Conditional Execution

- After

```
CMP r3 ,r4
ADDEQ r0,r1,r2 ; f = g + h (skipped if i ≠ j)
SUBNE r0,r1,r2 ; f = g - h (skipped if i = j)
```

- No branches
- Increases performance

ARM instruction format summary



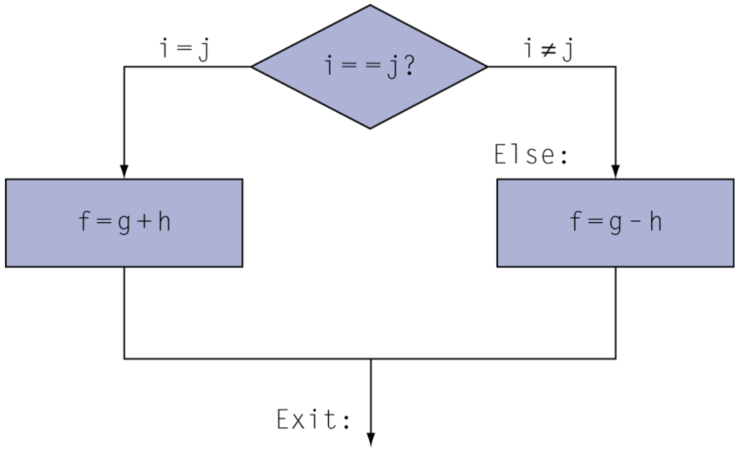
Name	Format	Example								Comments
Field size		4 bits	2 bits	1 bit	4 bits	1 bit	4 bits	4 bits	12 bits	All ARM instructions are 32 bits long
DP format	DP	Cond	F	I	Opcode	S	Rn	Rd	Operand2	Arithmetic instruction format
DT format	DT	Cond	F	Opcode			Rn	Rd	Offset12	Data transfer format
Field size		4 bits	2 bits	2 bits	24 bits					
BR format	BR	Cond	F	Opcode	signed_immed_24					B and BL instructions

Compiling If Statements

- C code:

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

  - *f,g,h,i,j in r0,r1,r2,r3,r4*



- Compiled ARM code:

```
        CMP r3,r4
        BNE Else                   ; go to Else if i != j

        ADD r1,r1,r2               ; f = g + h (skipped if i != j)
        B Exit
Else :

        SUB r0,r1,r2               ; f = g - h (skipped if i = j)

Exit:
```

Assembler calculates addresses

Exercise 3

- Use the template provided as ex3.s to convert the following C code to assembly

```
if (i>=j) f = a+b;
else f = a-b;
```

  - Use signed comparison
  - a,b,i,j in r0,r1,r2,r3 respectively
  - Put f to r5

Show your work to an instructor

Exercise 4

- Use the template provided as ex4.s to convert the following C code to assembly

```
if (i>5) f = 70;
else if (i>3) f=55;
else f = 30;
```

  - i in r0
  - Put f to r5
  - Hint : Use MOV instruction
    - MOV r5,#70 makes r5=70

Show your work to an instructor

Compiling Loop Statements

- C code:

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

  - *i in r3, k in r5, base address of save in r6*

- Compiled ARM code:

```
Loop:  ADD r12,r6, r3, LSL # 2  ; r12 = address of save[i]
       LDR r0,[r12,#0]          ; Temp reg r0 = save[i]
       CMP r0,r5
       BNE Exit                 ; go to Exit if save[i] ≠ k
       ADD r3,r3,#1             ; i = i + 1
       B Loop                   ; go to Loop
Exit:
```

Exercise 5

- Use the template provided as ex5.s to convert the following C code to assembly

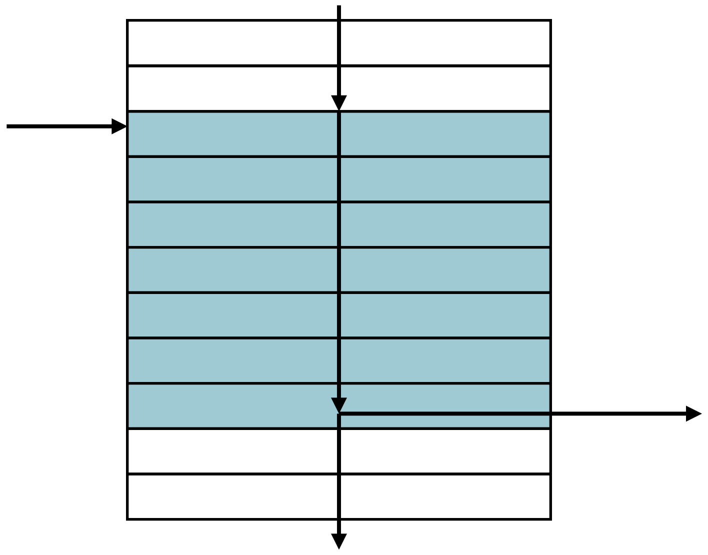
```
j=0;
for (i=0;i<10;i++){
    j+=i;
}
```

  - Put final j to r5

Show your work to an instructor

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