ECE222

Andy Zhang

Fall 2014

Contents

Chapter 1

Computers

1.1 Classes (1.1)

1.1.1 Personal

- Desktop, laptop, tablet, smartphones
- %1 of all CPUs sold(10 billion in 2008)
- Cost: $20 - 200

1.1.2 Embedded

- Integrated into a larger device or system
  - Automotive(airbags, ABS, ...)
  - Appliances: stove, microwave...
  - Airplanes
- 99% of all CPUs
- Cost: Microchip PIC12: $0.41

1.1.3 Servers

- Provides service to many users
  - Cloud computing (Amazon EC2, Azure ...)
  - Mainframes (IBM System Z) used by banks, universities, governments due to high reliability
  - Supercomputers — weather modelling, protein folding..
- <1% of all CPUs sold
- cost: $2000 / chip

1.2 Structure (1.2)

Definition : a computer is a 'programmable device that can store, retrieve and process data' — Merriam Webster

Computers of all classes can be decomposed into five types of functional events

1. Input: Mouse, Punchard, Touch Screen, Camera
2. Output: Printer, Screems.
3. Storage: Data, instructions (binary)
    - Memory is organized into a linear array of bytes
4. ALU: Arithmetic Logic Unit
    - Performs operations on data stored in registers
    - Add, multiply, AND, NOT, ...
5. Control Unit
    - Interpret instructions, fetch operands, control ALU

Chapter 2

Processors

2.1 Processor

PC program counter stores memory address of next instruction

IR instruction register stores instruction read from memory

MAR memory address to register outputs address to memory

MDR memory data register. Holds data/instructions from memory or going to memory

2.2 Instruction execution

2.2.1 Instruction Fetch (IF)

- Copy PC contents to MAR and assert R/$\bar{W}$ control signal
- Wait for response from memory and copy MDR contents to IR
- Increment PC

2.2.2 Instruction Decode (ID)

- Interpret bits in IR

2.2.3 Operand Fetch (OF)

- Read data from registers and/or extract constants from IR

2.2.4 Execute (EX)

- Use ALU or read memory (load) or write memory (store)

2.2.5 Writeback (WB)

Write result to a register

Eg Execute Load R2, LOC (memory address label)

1. Always same as above
2. Recognize 'Load'
3. Extract LOC from IR
4. Copy LOC to MAR and assert R/$\bar{W}$ control signal
5. Copy MDR Contents to R2

2.2.6 Homework

```
ADD R4, R2, R3 (R4 <- [R2] + [R3])
Store R4, LOC
```

2.3 Design Paradigms

2.3.1 CISC

Complex Instruction Set Computer

- Machine instructions can perform complex operations

  E.g. (x86) *movsb* copies an array of bytes
- Instructions are variable length
- Operands come from registers or memory

  E.g *M68K* ADD DO, LOC (mem[LOC] <- [DO] + [mem[LOC]])
- Complex addressing modes

  E.g. (M68K) ADD DO, (A0)+
- Smaller object code
- Direct support of High Level Language constructs
- Ease of assembly language programming
- Hardware is difficult to pipeline (speed up)

2.3.2 RISC

Reduced instruction-set computer

- Fewer, simpler instructions
- Load/store architecture
  - only load or store
  - ALU operands only come from registers

Eg (ARM)

```
ldr r1, LOC
add r1, r0, r1
ldr r2=LOC
str r1, [r2]
```

- Object code is larger (by ∼%30)
- Hardwire easier to pipeline

2.4 Register Transfer Notation

(no standard)

- Expresses the semantics of instruction execution as data transfers and control flow (logic)
- Memory locations are assigned labels e.g. LOC, A
- Registers are named R0, R1, PC, IR

x denotes contents of x

E.g.

- [LOC] contents of memory at LOC
- [R0] contents of register R0
- [[R0]] contents of memory at the location specified by contents of R0

',' denotes parallel
';' dnotes sequential

E.g. ADD R4, R2, R3

R4 <– [R2] + [R3]

E.g. instruction fetch

MAR <– [PC], R/$\bar{W}$ <– 1, PC <– [PC] + 4
IR <= [MOR]

2.5 Memory

- A processor can access a finite amount of physical memory, determined by the # of address pins
- Memory is measured in binary units but reported with S.I. prefixes (e.g. 1kB = 1024 bytes)
- Hard disks are measured in decimal units (e.g. 1kB = 1000 bytes)
- IEC introduced binary units to eliminate confusion (e.g. 1kiB = 1024 bytes)

Endianness

- Big endian: MSB (most sig. byte) is at low address, LSB at high address
- Little-endian: MSB at high address, LSB at low address

Ex: 1234AB**CO**

Big Endian		Little Endian	
0	12	0	CO
1	34	1	AB
2	AB	2	34
3	CO	3	12

Chapter 3

ARM

3.1 Background

- Acon/Adoned RISC Machines
- License designs to other companies to manufacture
- Target low power/low cost

Documentation: `http://infocenter.arm.com/help/index.jsp`

3.2 Design Principles

RISC but with some CISC characteristics
RISC

- Fixed instruction size
- load/store architecture

CISC

- Autoincrement/decrement addressing modes
- Move multiple values from registers to memory, in 1 instruction
- Condition codes

3.3 Memory

- Data sizes:
  - Word = 32 bits
  - Half word = 16 bits
  - Byte = 8 bits
- Word addresses are 'word aligned' (multiple of 4)
- Little or big endian
- Loads of half words or bytes at 200 extended or sign extended to 32 bits

3.4 Registers

Registers

- All registers are 32 bits

- 13 General purpose registers R0-R12, and
  R13 is the stack pointer (SP)
  R14 is the link register (LR)
  R15 is the program counter (PC)
- Condition code flags

R28 (V) Overflow
R29 (C) Carry out
R30 (Z) Zero
R31 (N) Negative

- Program status register

3.5 Instruction Set

3.5.1 Variations

3 variations

- ARM — 32 bit Thumb — 16 bit (compact, limited instructions, 8 operands) Thumb 2 — Mix of 16 and 32 bit, Cortex M3 in lab

3.5.2 Data Processing Instructions

Most have this format

```
<op>{flags}{cond} Rd, Rn, Op2
```

Op2 Operand 2 (right operand)
Rn Source register (Left operand)
Rd Destination register
{cond} Execute if condition is true
{flags} E.g. S => set condition code flags
< op> Operation meumonic

```
ADDEQ R2, R0, #1
if Z==1, R2 <- [R0] + 1
```

Operand 2

- an 8 bit constant (optionally rotated)
- Register value (Rm) optionally shifted

LSL Logical Shift Left, shift all bits to the left with 0 as LSB (multiply)
LSR Logical Shift Right, shift all bits to the right with 0 as MSB (unsigned divide by two)
ASR Arithmetic Shfit Right, shift all bits to the right, MSB becomes itself (signed divide by two)
RDR Rotate right
RRX Rotate right extend

```
ADD r2, r0, r1, LSL #2
r2 <- [r0] + [r1] << 2
= r2 <- [r0] + 4*[r2]
```

Arithmetic ADD, ADC (add with carry), SUB, SBC, RSB (reverse subtract)
Logical (bitwise) AND, ORR, EOR, BIC (and not, bitwise clear), ORN

```
(or not), (no NOT) ---- EOR Rd, Rn \#0xffffff
```

```
ADD R2, R0, #1
ADD R2, R0, R1, LSL, #2
```

- R_d => $R2$
- R_n => $R0$
- $op2$ => #1 and $R1, LSL, \#2$

3.6 D.P. Instructions

3.6.1 Data movement

```
MOV{S}{cond} Rd, Op2
MOV{cond}    Rd, #imm16 <---- zero extended to 32 bits
MVN{S}{cond} Rd, Op2    ; move not
MOVT{cond}   Rd, #imm16
```

E.g. move $0xabcd1234$ into $R0$

```
MOV R0, #0x1234
MOVT R0, #0xabcd
```

```
MVN R0, #1
R0 <---- NOT 1
= R0 <---- 0xfffffffe
```

3.6.2 Shift

```
format. <op>{S}{cond} Rd, Rm, <Rs|#n>
<op>: ASR, LSL, LSR, RGR, RRX
```

```
LSL R1, R0, R2 <---- contain n
// R1 <- [R0] << n
```

3.7 Memory Access Instructions

```
<op>{size}{cond} Rd, <address>
```

address Register indirect addressing
Rd Destination reg (loads), source reg (stores)
cond Only execute if true
size B = byte, H = halfword, SB = Signed byte, SH = Signed half word (sign extended to 32 bits on load). Size defaults to word
op LDR (load register) or STR (store register)

With immediate offset:

```
<address> = [Rn {, #offsets}]
```

```
LDR R1, [R0]
// r1 <- [[R0] + 0]
```

With register offset

```
<address> = [Rn, Rm{, LSL #n}]
```

```
LDR R1, [R0, -R2]
// R1 <- [[R0] - [R2]]
LDR R1, [R0, R2, LSL #2]
// R1 <- [[R0] + [R2] << 2]
```

3.7.1 Sum Array

See the demo0.s sheet.

3.8 Memory

3.8.1 Definitions

DCD = Declare Data

```
DCD 5
DCB 1, 2, 3, 4, 5
```

This is very similar to declaring

```
const int N = 5
const int ARRAY[] = [1, 2, 3, 4, 5]
```

ADR = Put address in register

```
int *ptr = ARRAY,
int total = ptr[0];
int a = ptr[1];
total += a;
```

Example
pc relative: LDR{size}{cond} Rt, label

- label is translated into an offset from the instruction
- data is loaded from <address>= PC + 4 + offset
- offset ∈ [[pc, #28]]

```
LDR r1, DATA1 ; DATA 1 is 322 bytes after LDR
=>
LDR r1, [pc, #28] ; pc incremented by 4 during fetch
// r1 <- [[pc] + 28]
```

load address into register: ADR{cond} Rd, label

```
ADR r1, DATA1
// r1 <- [pc] + 28
```

3.9 Pseudo Instructions

- don't match an existing machine instruction
- translated by the assembler into an appropriate instruction(s)

```
LDR {cond} Rt = <expr> ; label or numeric expression at end
```

- converted into
  - MOV or MVN, if possible, or
  - adds the value to the literal pool (program contants, bottom of file) and generates a LDR instruction with pc + relative addressing from demo0.s file

    ```
    LDR r1 = SUM
    ```

  - couldn't use ADR because the distance from LDR (0x114) to sum (0x1000 0000) exceeds ±4095
- Stores 0x1000 0000 at address 0x134 and replaces it with:

```
LDR r1, [pc, #28]
```

3.10 Branch Instruction

- Changes control flow by adding an offset to the PC
- Format: B{cond} label
  cond only execute if condition is true
  label assembler replaces with pc relative offset
- condition code suffixes (SIGNED)
  EQ equal (to zero)
  NE not equal
  GT greater than
  GE greater or equal
  LT less than
  LE less or equal
  PL plus (>= 0) ignores overflow
  MI minus (< 0) ignores overflow
- condition code suffixes (UNSIGNED)
  VS overflow set
  VC overflow clear
  Al always (default)

E.g. SUBS r2, r2, #1 // r2 ¡- [r2] - 1

```
BGT LOOP
do { int a = *ptr;
     total += a;
     ptr++;
     counter--;
   } while (counter > 0);} // demoX.s
```

3.11 ARM

3.11.1 Pre and Post indexed Addressing

- applies to LDR and STR
- pre indexed: <op>{size}{cond} Rt, [[Rn, #offset]]
  - the offset is added to the address in Rn, then the memory acess is performed and Rn is updated

```
LDR r1, [r0, #4]!
// r1 <- [[r0] + 4], r0 <- [r0] + 4
```

- post indexed: <op>{size}{cond} Rt, [[Rn]], #offset
  - the memory access is performed with the address in Rn, then Rn is updated by adding the offset

```
LDR r1, [r0], #4
// r1 <- [[r0]], r0 <- [r0] + 4
```

3.12 Compare Instructions

- Compares two operands and sets the condition flags(N, Z, C, V) but does not save to a destination register

- CMP {cond} Rn, Operand2. Example:

```
CMP r1, #1
// N, Z, C, V <= [r1] - 1
if r1 = 1, N <- 0, Z <- 1, C <- 1, V <- 0
  but r1 is not changed
```

- CMN{cond} Rn, Operand2 // compare negative
  - compares [[Rn]] and —Operand2
- test: TST{cond} Rn, Operand2
  - perform bitwise AND and updates flags N, Z, C, V. E.g. :

```
TST r1, \#0x00008000
// N, Z, C, V <= [r1] * 2_0000 0000 0000 0000 1000 0000 0000 000
if r1 = 0xF000 -> 0000 0000 0000 0000 1111 0000 0000 0000
                  0000 0000 0000 0000 1000 0000 0000 0000
```

- test equal: TEQ{cond} Rn, Operand2
  - performs bitwise EOR and updates flags
- compare and branch: <op>Rn, label. For op:

CBZ compare equal to zero
CBNZ compare not equal to zero

- compares [[Rn]] with zero and decides on branch

CBZ Rn, label =
```
    CMP Rn, #0
    BEQ label
```
CBNZ Rn, label =
```
    CMP Rn, #0
    BNE label
```

3.13 If Else

```
if (x == 0) // cond
  y++; // stmt 1
else
  y--; // stmt 2
x in r0, y is r1
```

This is equivalent to

```
        CBZ r0, ADD_ // cond
        SUB r1, r1, #1 // stmt 2
        B END
ADD_    ADD r1, r1, #1 // stmt 1
END_    // end code
```

Without branching:

```
CMP r0, #0
ADDEQ r1, r1, #1
SUBNE r1, r1, #1
```

3.14 Subroutines

3.14.1 The Stack

- LIFO
- Each thread/process has a call stack growing from high to low memory address
- Typical memory map (32 bit addressing)
- r13(alias sp) is the stack pointer
- Push[[r0]] onto the stack

```
STR r0, [sp, #-4]! // \! == update/write, (pre-indexed)
```

- Pop from stack into r0

```
LDR r0, [sp], #4 // post indexed
```

3.14.2 Calling

- Branch to subroutine instructions store return address in the link register r14 (alias lr)
- Branch and link: BL{cond} label (invocation)

```
// pc <- [pc] + offset // offset in [-16MB, 16MB]
r14 <- [return address] // pc of the next instruction
```

- Branch, link and exchange: BLX cond Rn (invocation)

```
eg: BLX r1
// pc <- [r1], r14 <- [pc]
// Greather range than BL [-2GB, 2GB]
```

- Branch exchange: Bx cond Rn (return)

```
e.g. Bx lr
// pc <- [r14]
```

3.14.3 Load/Store Multiple

- Push/pop multiple register values to/from stack
- Syntax < op> {mode}{cond} Rn{!}, reglist

  op = LDM load muliple, or SDM store multiple
  mode = IA increment after, or DB decrement before
  ! = writeback (update the stack pointer)
  reglist = comma separated list of regs or reg ranges
- STMFD is a synonym for STMDB (push)
- LOMFD is a synonym for LDMIA

  FD = full descending stack

```
e.g.\ push r4, r5, r6 and r14 onto the stack
STMFD sp!, [r4, r6, lr]
r4   | 0000 0004
r5   | 0000 0005
46   | 0000 0006
r13  | 1000 0200 <- sp
r14  | 0000 0100 <- lr
Memory after pushing
1000 01F0 | 0000 0004 <- sp
1000 01F4 | 0000 0005
1000 01F8 | 0000 0006
1000 01FC | 0000 0100
1000 0200 | prev valu <- previous sp

e.g.\ pop from stack into r4, r5, r6, pc
LDMFD sp! {r4——r6, pc}
```

3.14.4 AAPCS (Arm Architecture Procedure Call Standard)

registers	synonyms	callee preserved	function
r0-r3	a1-a4	no	argument/result/scratch regs
r4-r1	v1-v8	yes	local variable
r12	p	no	intro procedure/scracthing
r13	sp	yes	stack pointer
r14	lr	no	link register
r15	pc	no	program counter

Guidelines

- Preserve and restore v1 — v8 (r4 — r11) if you modify it
- Anything pushed in the stack must be popped/removed
- Return values are in r0 (and r1 — r3 as needed)
- Pass parameters via registers first (faster)
- pass additional parameters via stack