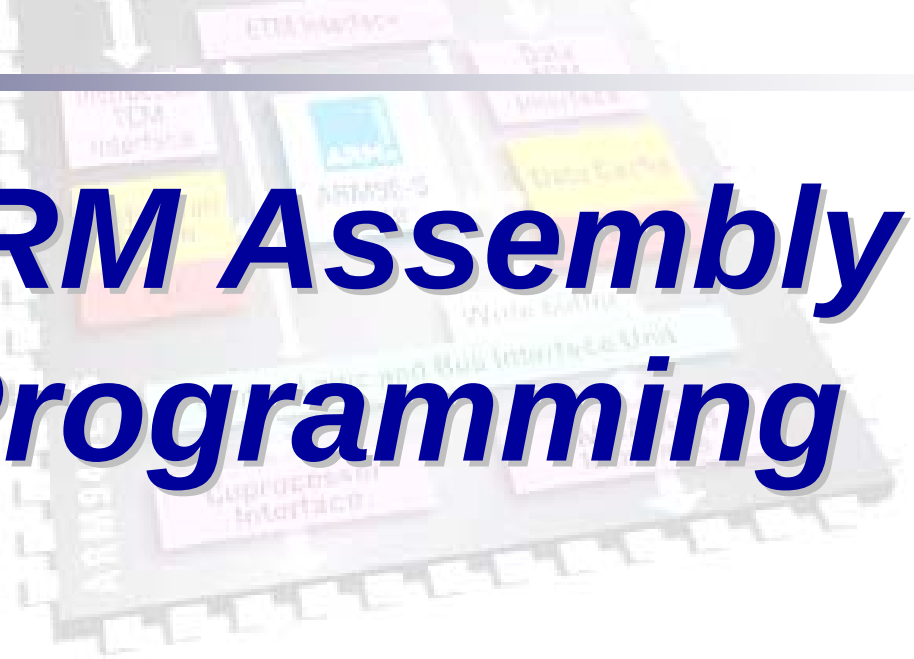
CEG 2400 Tutorial 4



ARM Assembly Programming

Instruction Set

- Two instruction sets:
  - ARM
    - Standard 32-bit instruction set
  - THUMB
    - 16-bit compressed form
    - Code density better than most CISC
    - Dynamic decompression in pipeline

ARM Instruction Set (1)

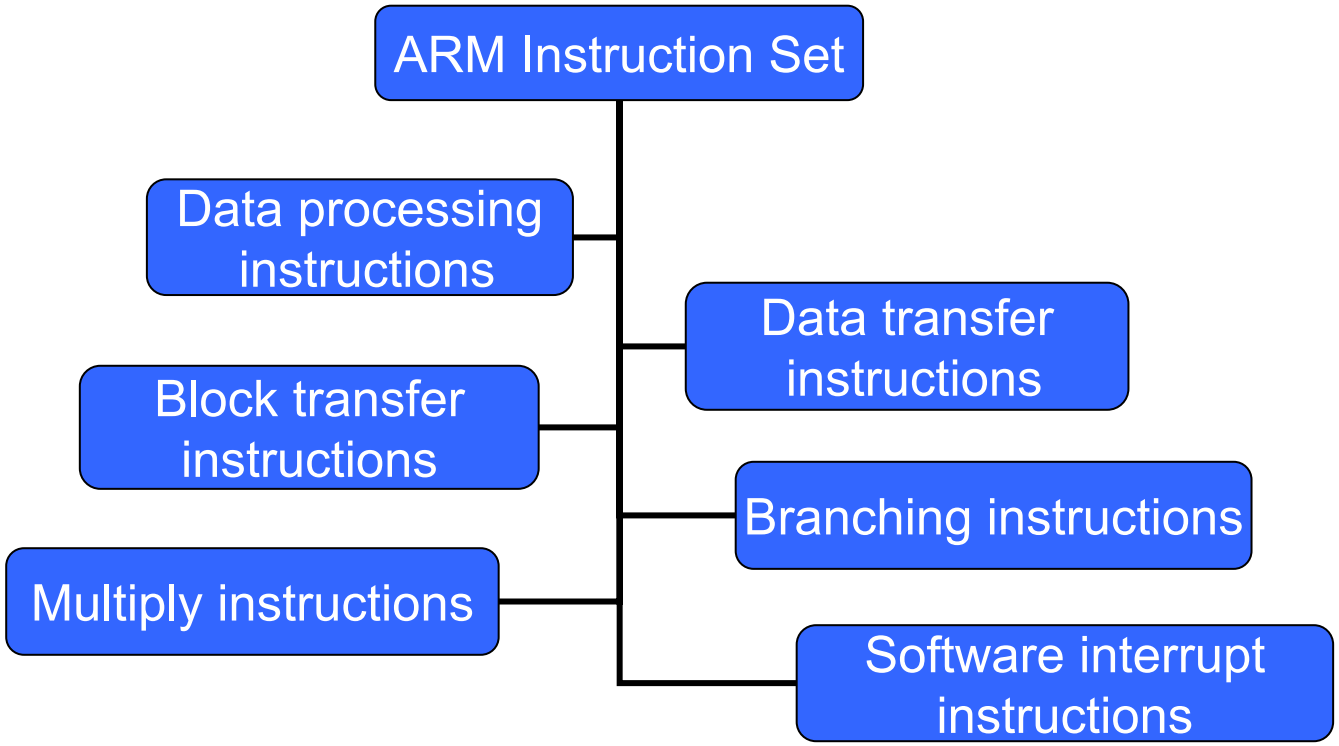
- Features:
  - Load/Store architecture
  - 3-address data processing instructions
  - Conditional execution
  - Load/Store multiple registers
  - Shift & ALU operation in single clock cycle

ARM Instruction Set (2)

- Conditional execution:
  - Each data processing instruction postfixed by condition code
  - Result – smooth flow of instructions through pipeline
  - 16 condition codes:

EQ	equal	MI	negative	HI	unsigned higher	GT	signed greater than
NE	not equal	PL	positive or zero	LS	unsigned lower or same	LE	signed less than or equal
CS	unsigned higher or same	VS	overflow	GE	signed greater than or equal	AL	always
CC	unsigned lower	VC	no overflow	LT	signed less than	NV	special purpose

ARM Instruction Set (3)



Data Processing Instructions (1)

- Arithmetic and logical operations
- 3-address format:
  - Two 32-bit operands
    (op1 is register, op2 is register or immediate)
  - 32-bit result placed in a register
- Barrel shifter for op2 allows full 32-bit shift within 1 instruction cycle

Data Processing Instructions (2)

- Arithmetic operations:
  - ADD, ADDC, SUB, SUBC, RSB, RSC
- Bit-wise logical operations:
  - AND, EOR, ORR, BIC
- Register movement operations:
  - MOV, MVN
- Comparison operations:
  - TST, TEQ, CMP, CMN

Data Processing Instructions (3)

- Data processing instructions + Conditional codes + Barrel shifter = Powerful tools for efficient coded programs
- E.g.:

if (z==1) R1=R2+(R3*4)

compiles to

ADDEQS R1,R2,R3, LSL #2

(SINGLE INSTRUCTION !)

Data Transfer Instructions

- Load/store instructions
- Used to move signed and unsigned
  Word, Half Word and Byte to and from registers
- Can be used to load PC
  (if target address is beyond branch instruction range)

LDR	Load Word	STR	Store Word
LDRH	Load Half Word	STRH	Store Half Word
LDRSH	Load Signed Half Word	STRSH	Store Signed Half Word
LDRB	Load Byte	STRB	Store Byte
LDRSB	Load Signed Byte	STRSB	Store Signed Byte

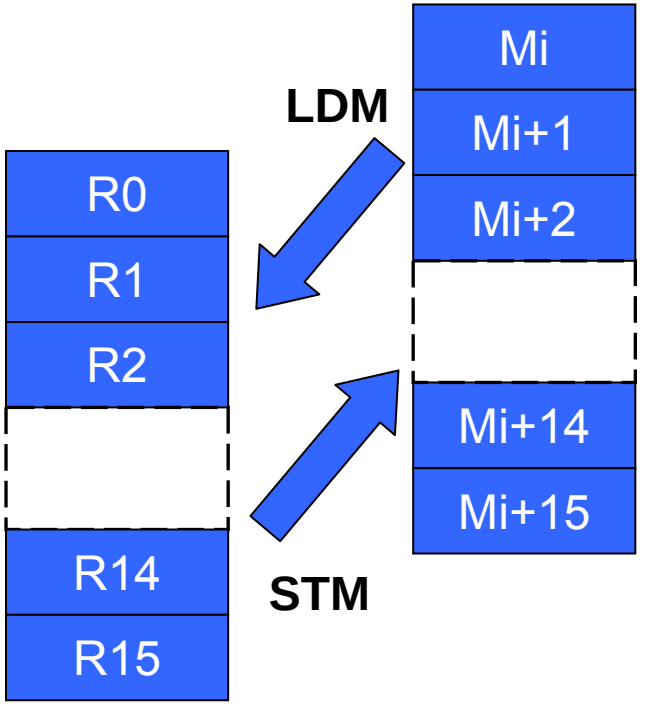
Block Transfer Instructions

- Load/Store Multiple blocks (*LDM*/*STM*)
- Whole register bank or a subset copied to memory or restored with single instruction

E.g.

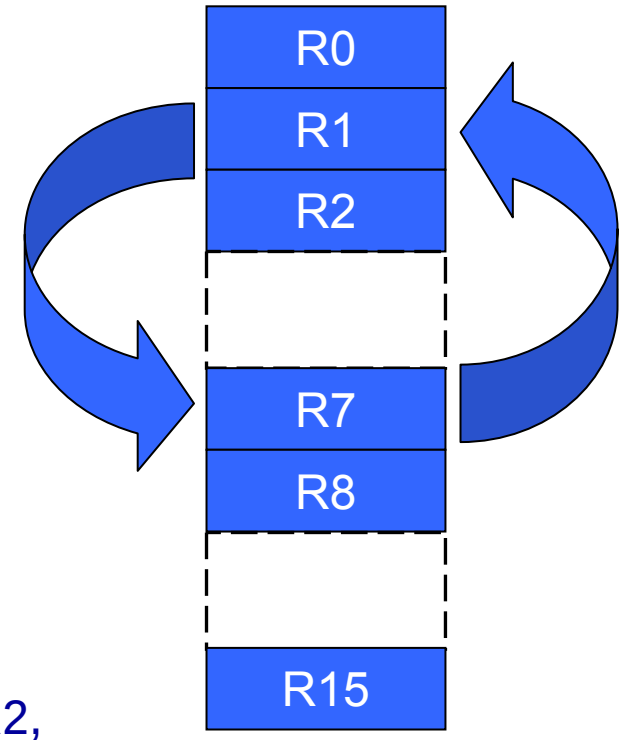
STMED SP!,{R0-R3,R14} @save R0 to R3 to use as workspace and R14 for retruning.

LDMED SP!,{R0-R3,R15} @ restore workspace and return



Swap Instruction

- Exchanges a word between registers
  - Two cycles but single atomic action
  - Support for RT semaphores



E.g.

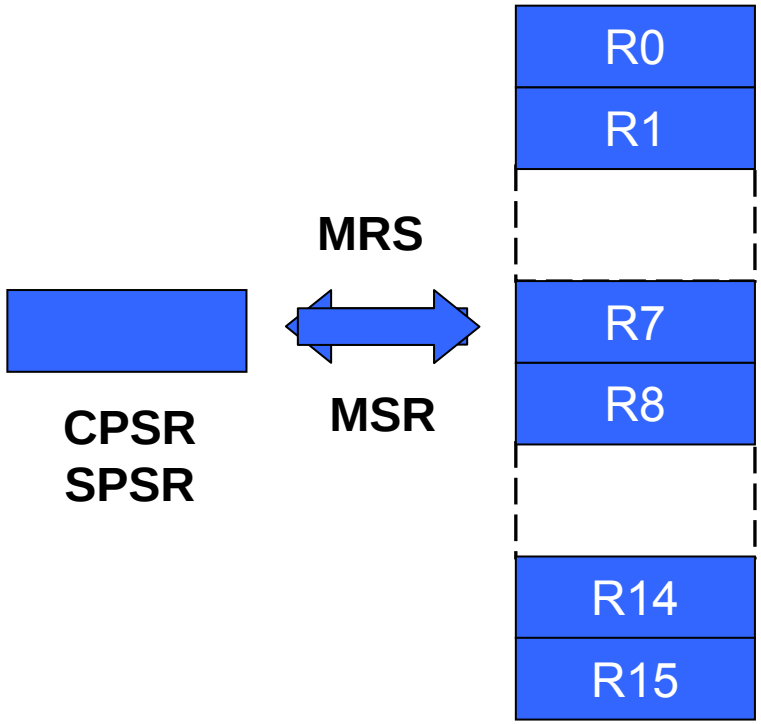
SWP R0,R1,[R2]

@Load R0 with the word addressed by R2, and store R1 at R2

SWPB R2,R3,[R4]

Modifying the Status Registers

- Only indirectly
- *MSR* moves contents from CPSR/SPSR to selected GPR
- *MRS* moves contents from selected GPR to CPSR/SPSR
- Only in privileged modes can change all valid bits



Multiply Instructions (1)

- Integer multiplication (32-bit result)
- Long integer multiplication (64-bit result)
- Built in Multiply Accumulate Unit (MAC)
- Multiply and accumulate instructions add product to running total

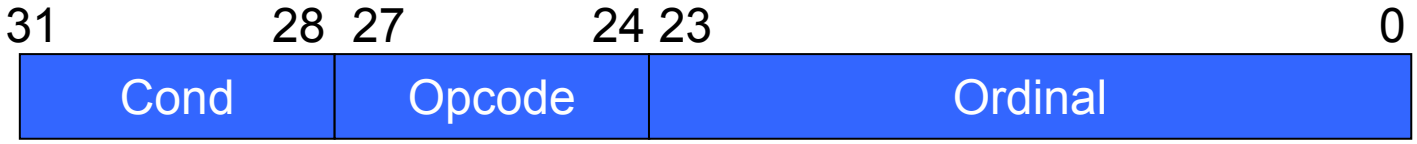
Multiply Instructions (2)

- Instructions:

MUL	**Multiply**	**32-bit result**
MULA	**Multiply accumulate**	**32-bit result**
UMULL	**Unsigned multiply**	**64-bit result**
UMLAL	**Unsigned multiply accumulate**	**64-bit result**
SMULL	**Signed multiply**	**64-bit result**
SMLAL	**Signed multiply accumulate**	**64-bit result**

Software Interrupt

- *SWI* instruction
  - Forces CPU into supervisor mode
  - Usage: SWI #n



  - Maximum 2^{24} calls
  - Suitable for running privileged code and making OS calls

Branching Instructions (1)

- *Branch* (B):
  jumps forwards/backwards
  up to 32 MB
- *Branch link* (BL):
  same + saves (PC+4) in LR
- Suitable for function call/return
- Condition codes for conditional branches

Branching Instructions (2)

- *Branch exchange (BX)* and
  Branch link exchange (BLX):
  same as *B/BL* +
  exchange instruction set (ARM ↔THUMB)
- Only way to swap sets

THUMB Instruction Set

- Compressed form of ARM
  - Instructions stored as 16-bit,
  - Decompressed into ARM instructions and
  - Executed
- Lower performance (ARM 40% faster)
- Higher density (THUMB saves 30% space)
- Optimal –
  "*interworking*" (combining two sets) –
  compiler supported

THUMB Instruction Set (2)

- More traditional:
  - No condition codes
  - Two-address data processing instructions
- Access to R0 – R8 restricted to
  - *MOV*, *ADD*, *CMP*
- PUSH/POP for stack manipulation
  - Descending stack (SP hardwired to R13)

THUMB Instruction Set (3)

- No *MSR* and *MRS*,
  must change to ARM to modify CPSR
  (change using *BX* or *BLX*)
- ARM entered automatically after
  RESET or entering exception mode
- Maximum 255 SWI calls

Bad Program *vs.* Good Program (1)

- Euclid's Greatest Common Divisor (gcd) algorithm
  - In C program language

```
int gcd (int a, int b)
{
        while (a!=b) do
        {
                if (a>b)
                  a=a-b;
                else
                  b=b-a;
        }
        return a;
}
```

  - In Arm Assembly language

```
gcd CMP   r0,r1
    BEQ   end
    BLT   less
    SUB   r0,r0,r1
    B     gcd
less
    SUB   r1,r1,r0
    B     gcd
end
```

Bad Program *vs.* Good Program (2)

- Euclid's Greatest Common Divisor (gcd) algorithm

Bad:

```
gcd CMP   r0,r1
    BEQ   end
    BLT   less
    SUB   r0,r0,r1
    B     gcd
less
    SUB   r1,r1,r0
    B     gcd
end
```

Good:

```
gcd CMP   r0,r1
    SUBGT r0,r0,r1
    SUBLT r1,r1,r0
    BNE   gcd
end
```

- Only four instructions
- Execute faster in most cases

Bad Program *vs.* Good Program (3)

- Conditional branches only

r0:a	r1:b	Instruction		Cycles(ARM7)
1	2	CMP	r0,r1	1
1	2	BEQ	end	1(not executed)
1	2	BLT	less	3
1	2	SUB	r1,r1,r0	1
1	1	B	gcd	3
1	1	CMP	r0,r1	1
1	1	BEQ	end	3
				Total=13

Bad Program *vs.* Good Program (4)

- All instructions conditional

r0:a	r1:b	Instruction	Cycles(ARM7)
1	2	CMP r0,r1	1
1	2	SUBGT r0,r0,r1	1 (not executed)
1	1	SUBLT r1,r1,r0	1
1	1	BNE gcd	1 (not executed)
			Total=4

GCC inline assembly (1)

- A simple Example:

```
// Passing Input parameters and return output varaible using assembler
int main(void) {

int i= 12, j =7;
int a;

// this assembly issue: a = i+ j statement
asm volatile( " add %0,%1,%2"
:"=r" (a) // a = %0
:"r" (j), "r" (i) // j = %1, i = %2
// no clobber list, so this line is empty
);

a = a + 9;
printf("a = %d \n",a);
return 0;
}
```

GCC inline assembly (2)

- The general form of an asm() is:

asm volatile("code":outputs:inputs:clobbers);

- Within the "code", %0 refers to the first argument (usually an output, unless there are no outputs), %1 to the second, and so forth. It only goes up to %9.
- Including multi-line asm, you should separate lines with "\n\t" or use ";" as a separator to put more than one asm on a line. Or just put these codes in different line
- Each output or input in the comma-separated list has two parts, "constraints" and (value). The (value) part is an expression. For outputs, it must be an lvalue.
- In "constraints", all outputs must be marked with "=".

GCC inline assembly (3)

- Use cross-compiler to compile this program .

  >arm-elf-gcc –elf2flt –o test test.c –static

  >file test

  >test: BFLT executable - version 4 ram

- Download the executable binary file to our arm simulator.

  >ftp 10.0.0.2

  >binary

  >get test

  >bye

- Execute this program in our arm simulator

Useful Web Addresses

- ARM
- http://www.arm.com/
- ARM Limited *ARM Architecture Reference Manual*, Addison Wesley, June 2000
- Trevor Martin *The Insiders Guide To The Philips ARM7-Based Microcontrollers*, Hitex (UK) Ltd., February 2005
- Steve Furber *ARM System-On-Chip Architecture (2nd edition)*, Addison Wesley, March 2000