EEE3096S: Embedded Systems II

LECTURE 12:

ARM ASSEMBLY PROGRAMMING (PART 1)

Presented by:

Dr Yaaseen Martin

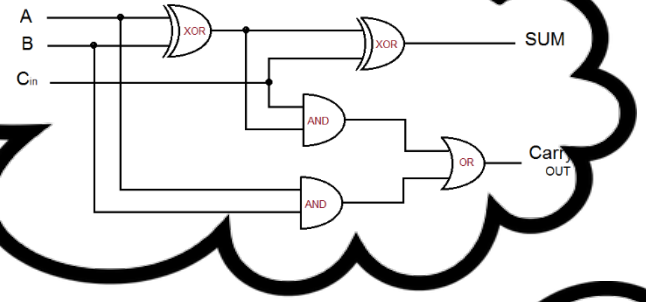
Electrical Engineering
University of Cape Town

CPU INSTRUCTIONS = DIGITAL LOGIC

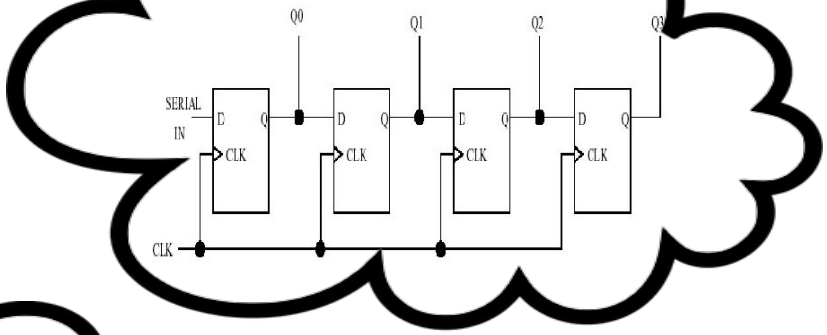
- Reflect back on digital logic experience
- You mind might be sparking ideas of e.g.

...

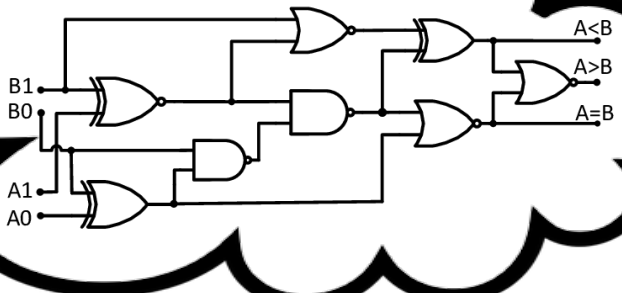
ADDER



SHIFT REGISTER



COMPARATOR



PSEUDO-ASSEMBLY

- Thinking about the building blocks of CPU instructions
- This “learning Assembly using pseudo-Assembly” approach is a two-pass process…
- Here is a first pass:

Thinking about instructions to use, and how to order them, to run a program.
(Note that we have not yet defined the instructions! But that’s the point: think up your own ideas for them and their sequencing – that’s the approach)

PROGRAM EXAMPLE

We want to carry out the following processing:

```
int a, b, avg;
a = 100;
b = 200;
avg = (a+b)/2;
```

PSEUDO-CODE SOLUTION

Program wanted:

```
int a, b, avg;
a = 100;
b = 200;
avg = (a+b)/2;
```

0. Some registers

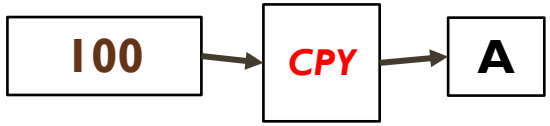
1. Assign these registers to values

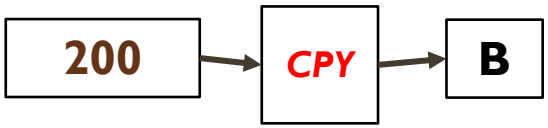
2. Assign avg to (a+b)

3. Assign avg to avg/2

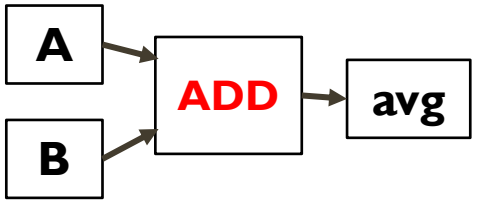
0. Some registers, let's say 32-bit ones

1. CPY A,100
   CPY B,200

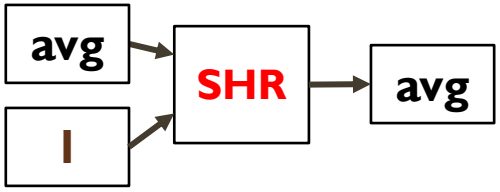




2. ADD AVG,A,B



3. SHR AVG, AVG, 1 // i.e. shift right = divide by 2 for ints



Thinking Activity Done

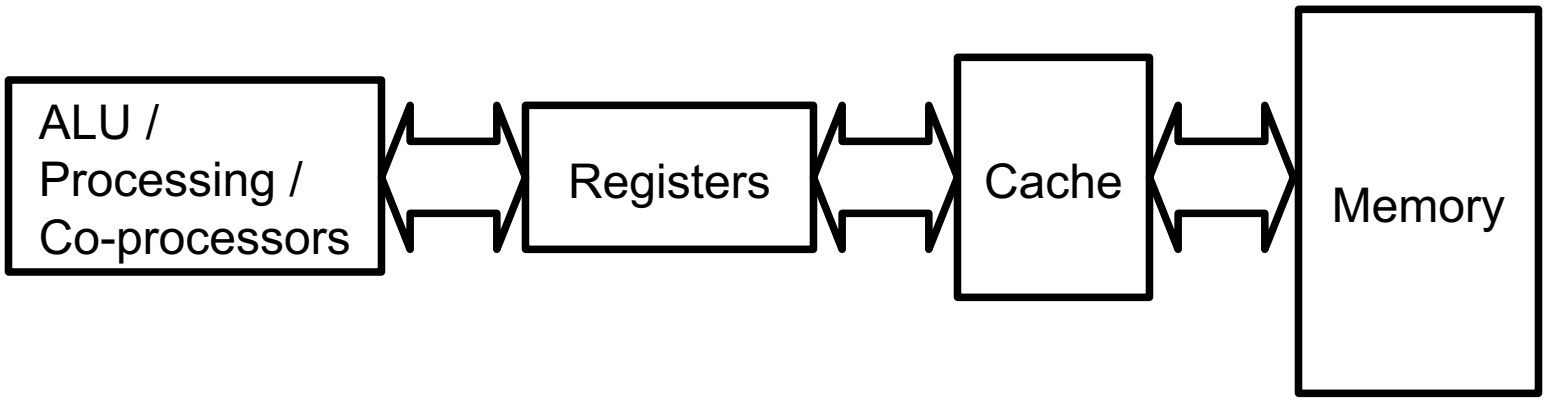
First pass of learning Assembly: making up your own realistic Assembly commands.

Hopefully you know the general approach now of writing Assembly (and may be convinced also why this approach is useful for designing processors)

Now for second pass of learning ARM Assembly specifically…

ARM DATA AND INSTRUCTIONS

- We will focus on the ARM AArch32 state; instructions and data are **32-bit**
- ARM uses Load/Store architecture – you cannot manipulate memory directly
- ARM is a **RISC** processor – the instruction set is not as rich as for the x86
- **Co-processors** can be used to supplement main processor's operations, e.g. floating-point computations
- **Cache** is a buffer between main memory and the rest of CPU; used for frequently-used instructions/data for quick access



ARM INSTRUCTION SET

Instruction set divided into six types:

1. Branch Instructions
2. Data Processing instructions
3. Status Register Transfer Instructions
4. Load and Store Instructions
5. Co-processor Instructions
6. Exception-generating Instructions

ARM Instruction Set

Branch

B	Branch
BX	Branch and Exchange
BL	Branch and Link

Data Processing

ADD	Add
SUB	Subtract
RSB	Reverse Subtract
CMP	Compare
TST	Test
AND	Logical AND
EOR	Logical Exclusive OR
MUL	Multiply
SMULL	Sign Long Multiply
SMLAL	Signed Long Multiply Accumulate
MOV	Move
SWP	Swap Word
MVN	Move Not
ADC	Add with Carry
SBC	Subtract with Carry
RSC	Reverse Subtract with Carry
CMN	Compare Negated
TEQ	Test Equivalence
BIC	Bit Clear
ORR	Logical (inclusive) OR
MLA	Multiply Accumulate
UMULL	Unsigned Long Multiply
UMLAL	Unsigned Long Multiply Accumulate
SWPB	Swap Byte

Status-register transfer functions

MSR	Move to Status Register
MRS	Move From Status Register

Load and Store

LDR	Load Word
LDRSH	Load Signed Halfword
LDRSB	Load Signed Byte
LDRH	Load Half Word
LDRB	Load Byte
LDRBT	Load Register Byte with Translation
LDRT	Load Register with Translation
LDM	Load Multiple
STR	Store Word
STRH	Store Half Word
STRB	Store Byte
STRBT	Store Register Byte with Translation
STRT	Store Register with Translation
STM	Store Multiple

Co-Processor

LDC	Load To Coprocessor
CDP	Coprocessor Data Processing
MRC	Move From Coprocessor
STC	Store From Coprocessor

Exception-generating

SWI	Software Interrupt

GNU ASSEMBLER (GAS)

GAS is the GNU Assembler, which comes with GNU **binutils** for a variety of platforms

Syntax for ARM Assembly instructions:

[label:] **<*instruction*>** **<*suffix*>** *[operands]*

e.g. add_function: ADDEQ, r0, r0, r1

Notes:

- *label* is optional – only needed if you want to refer to the instruction (e.g. to **branch** to the instruction)
- *suffix* can be used to implement **conditional operation** (e.g. EQ); reduces the need for branches and increases code density
- Not all instructions have operands
- Comments in Assembly start with **@**

CONDITIONAL EXECUTION

Suffix	Meaning
EQ	Equal
NE	Not equal
CS	Carry set (identical to HS)
HS	Unsigned higher or same (identical to CS)
CC	Carry clear (identical to LO)
LO	Unsigned lower (identical to CC)
MI	Minus or negative result
PL	Positive or zero result
VS	Overflow
VC	No overflow
HI	Unsigned higher
LS	Unsigned lower or same
GE	Signed greater than or equal
LT	Signed less than
GT	Signed greater than
LE	Signed less than or equal
AL	Always (this is the default)

Addition of Q flag in 64-bit state which indicates floating point saturation

BRANCH INSTRUCTIONS

B : Branch

• B <address> (e.g. B main)

• Address must be within -32 MB to +32 MB of the current instruction (i.e. ± 32 MB relative to the Program Counter)

• Does not return

BL : Branch and Link

• BL <subroutine> (e.g. BL sqrt)

• (Program Counter + 4 bytes) is saved into the Link Register so that PC will resume from **next** instruction when returning

• After subroutine runs, PC is restored using saved LR value

• To return, use ***bx lr*** to return (Branch and Exchange to the Link Register)

CONDITIONAL BRANCHING

- Branches can be **conditional** too
- The assembler equivalent of an *if* statement

B<suffix> <offset>

- Branch if suffix matches state of flag in the CPSR
- Examples:
  - **BEQ** : Branch if equal or zero
  - **BMI** : Branch if the result is negative
  - **BVS** : Branch if an overflow occurred

CONDITIONAL BRANCHING

Branch	Interpretation	Normal uses
B	Unconditional	Always take this branch
BAL	Always	Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC	Carry clear	Arithmetic operation did not give carry-out
BLO	Lower	Unsigned comparison gave lower
BCS	Carry set	Arithmetic operation gave carry-out
BHS	Higher or same	Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occurred
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Unsigned comparison gave lower or same

DATA PROCESSING INSTRUCTIONS

- Syntax **OP<suffix>** <result>, <op1>, <op2>

ADD : Add without carry

- result = op1 + op2

ADC : Add with carry, i.e., adds an extra “1” if the Carry flag is set

- result = op1 + op2 + carry

SUB : Subtract without carry

- result = op1 - op2

DATA PROCESSING INSTRUCTIONS

- If you want to use **flags**, you must specify that the status register must be updated
- Useful if you have a **condition check** in the next instruction, e.g., BEQ
- For example, to add two multi-word values:

ADDS R0, R4, R8 @ Add without carry

ADCS R1, R5, R9 @ Add with carry

ADC R1, R5, R9 @ No need to update flags here

DATA PROCESSING INSTRUCTIONS

- **AND**, **ORR**, **EOR** work similarly to **ADD**

For assignment, use **MOV**

- **MOV** r1, r2 @ r1 = r2 (assignment)
- **MVN** r1, r2 @ r1 = ~r2 (assignment with complement)

For comparison, use **CMP**

- **CMP** r1, r2 @ set condition flag on r1 – r2
- Same as **SUBS** but result is discarded

DATA PROCESSING: SHIFTING

- A shift can be applied to the second source register
- Can shift by a constant or by a register
- **LSR** logical shift right (**LSL** for left); treats number as **unsigned**
- **RRX** rotate right with carry (no left)
- **ROR** rotate right without carry (no left)
- **ASR** arithmetic shift right (no left); treats number as **signed**

e.g.:

ADD r1, r2, r3, **LSL** r4 @ r1 = r2 + r3*2^r4

STATUS REGISTER TRANSFER

Content of registers can be moved into and out of the status registers using MSR:

MSR Move to status register

- **MSR<suffix>** <sr>, <reg>

e.g. **MSR** cpsr, r10

MRS Move from status register

- **MRS<suffix>** <reg>, <sr>

e.g. **MRS** r10, cpsr

LOAD AND STORE

Content of registers are moved from/to memory using LDR/STR respectively:

LDR Load Register, e.g.:

- **LDR** r0, =jump @ Loads **address** of the label *jump* into r0
- **LDR** r1, =100 @ Loads **value** 100 into r1
- **LDR** r1, [r2] @ r1 = mem[r2]; loads **value** at **address** in r2 into r1

STR Store Register, e.g.:

- **STR** r1, [r2] @ mem[r2] = r1; stores **value** in r1 at address in r2

INDEX ADDRESSING

Index addressing can make load/store operations more efficient, i.e. it adds an offset to the base address as part of the same instruction.

Similar C statements using int arry[100], with each element being 4 bytes:

Pre-index addressing

```
int x = arry[i+1];
```

- **LDR** r1, [r2, #4] @ r1 = mem[r2+4]

Auto Pre-index addressing

```
int x = arry[++i];
```

- **LDR** r1, [r2, #4]! @ r1 = mem[r2+4]; r2 += 4

Auto Post-index addressing

```
int x = arry[i++];
```

- **LDR** r1, [r2], #4 @ r1 = mem[r2]; r2 += 4