EEE3096S: Embedded Systems II

LECTURE 13:

ARM ASSEMBLY PROGRAMMING (PART 2)

Presented by:

Dr Yaaseen Martin

Electrical Engineering
University of Cape Town

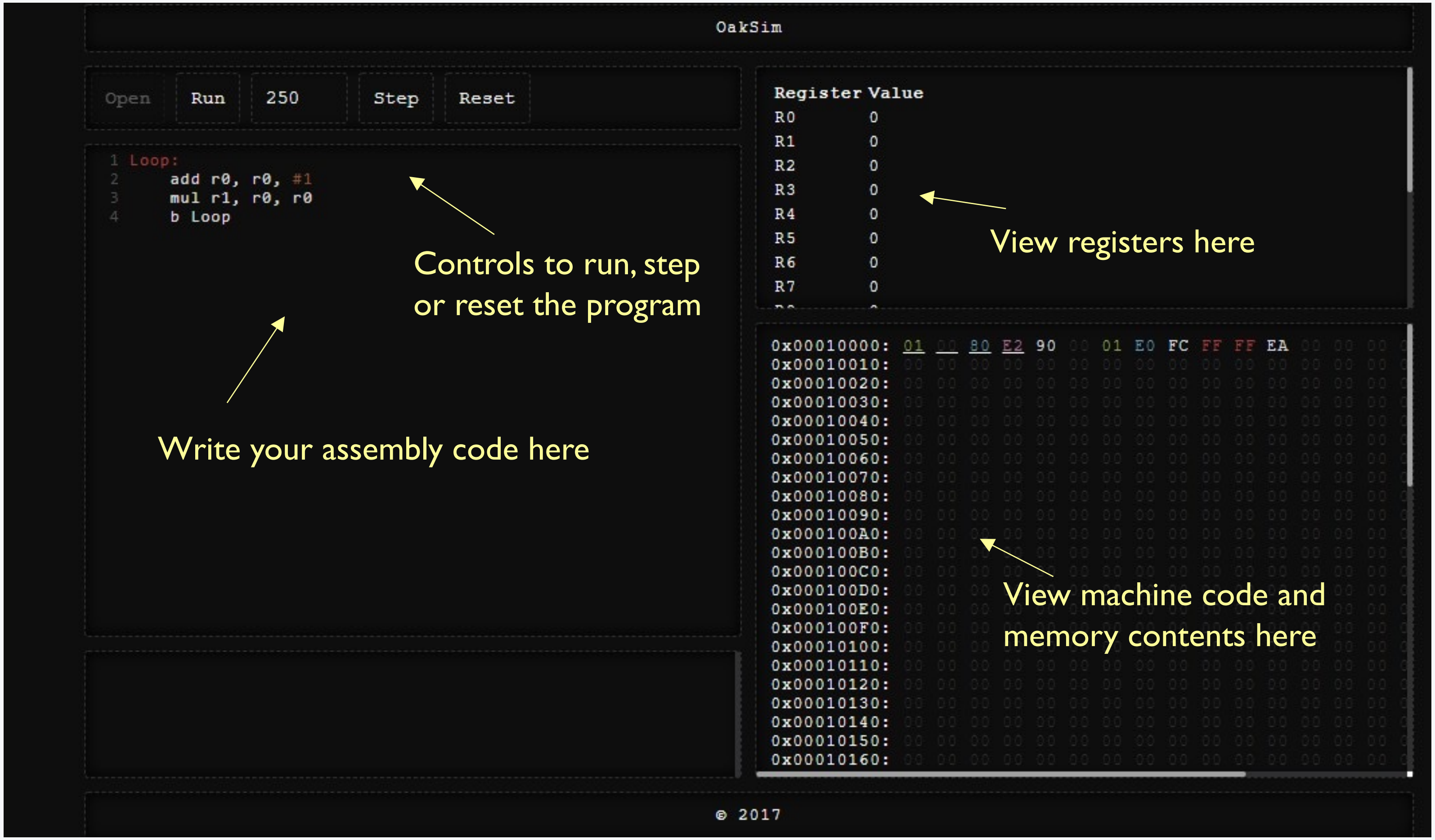
CPU INSTRUCTIONS

- As mentioned in last lecture: this Assembly training follows a two-pass approach…

- Pass 1:
  - Dreaming up our own Assembly code, thinking of instructions that may be useful, how they would look, etc.
- Pass 2:
  - With a basic understanding of ARM instructions and GAS coding, we now apply this by writing out own Assembly code that could run on an ARM processor

OAKSIM: ONLINE ARM SIMULATOR

OakSim is a very useful ARM AArch32 Assembly simulator:
https://wunkolo.github.io/OakSim/
Note: The usual file extension for ARM Assembly is .s



```
Loop:
    add r0, r0, #1
    mul r1, r0, r0
    b Loop
```

```
Register Value
R0        0
R1        0
R2        0
R3        0
R4        0
R5        0
R6        0
R7        0
```

```
0x00010000: 01 00 80 E2 90 00 01 E0 FC FF FF EA 00 00 00
0x00010010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00010020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00010030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00010040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00010050: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00010060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00010070: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00010080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00010090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000100A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000100B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000100C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000100D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000100E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000100F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00010100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00010110: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00010120: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00010130: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00010140: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00010150: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00010160: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Controls to run, step or reset the program

View registers here

Write your assembly code here

View machine code and memory contents here

ARM ASSEMBLY ACTIVITY

- For this activity you are tasked to “convert” the C code on the next slide into ARM Assembly.
- *TODO:*
  - Look at the next slide and contemplate the ARM instructions available, which have been covered in this lecture, and think which of them you would use.
  - (No need to fully solve the problem yet!)
  - Then proceed to the subsequent slide that gives a solution.

ACTIVITY PROBLEM DESCRIPTION

We want to carry out the following processing:

```
int a, b, avg, res;
a = 100;
b = 200;
avg = (a+b)/2;
if (a>avg)  res=1;  // a greater than avg
if (a<avg)  res=-1; // a less than avg
if (a==avg) res=0;  // a same as avg
```

ACTIVITY:
~5 MINUTES

Work with a classmate to convert to ARM Assembly:

```
int a, b, avg, res;
a = 100;
b = 200;
avg = (a+b)/2;
if (a>avg)  res=1;
if (a<avg)  res=-1;
if (a==avg) res=0;
```

OR

Fall asleep and wait for the solution:

ACTIVITY SAMPLE SOLUTION

Reference C code:

```
int a, b, avg, res;
a = 100;
b = 200;
avg = (a+b)/2;
if (a>avg)  res=1;
if (a<avg)  res=-1;
if (a==avg) res=0;
```

Step 1: Make a starting point for the procedure you are composing…

```
compaavg:               @ function compaavg that compares
                        @ param a to average of params a+b
```

Reference C code:

```
int a, b, avg, res;
a = 100;
b = 200;
avg = (a+b)/2;
if (a>avg)  res=1;
if (a<avg)  res=-1;
if (a==avg) res=0;
```

Step 2: We need to choose to registers, say r1 and r2 to hold the values for a and b

```
compaavg:          @ function compaavg that compares
                   @ param a to average of params a+b
  mov  r1, #100    @ set r1 = a = 100
  mov  r2, #200    @ set r2 = b = 200
```

Reference C code:

```c
int a, b, avg, res;
a = 100;
b = 200;
avg = (a+b)/2;
if (a>avg)  res=1;
if (a<avg)  res=-1;
if (a==avg) res=0;
```

Step 3: Now we need to find avg = (a+b) / 2. We are going to do the /2 by an arithmetic logic shift right instead of using a divide.

```
compaavg:            @ function compaavg that compares
                     @ param a to average of params a+b
  mov  r1, #100      @ set r1 = a = 100
  mov  r2, #200      @ set r2 = b = 200
  add  r3, r1, r2    @ avg = r3 = a + b
  asr  r3, r3, #1    @ avg = avg / 2
```

Reference C code:

```
int a, b, avg, res;
a = 100;
b = 200;
avg = (a+b)/2;
if (a>avg)  res=1;
if (a<avg)  res=-1;
if (a==avg) res=0;
```

Step 4: Compare a to avg, which will set the flags GT (if a>avg) or LT (if a<avg) or EQ (if a=avg); however, we only need two branches because the last option does not need a branch.

```
compaavg:               @ function compaavg that compares
  ...
  asr  r3, r3, #1  @ avg = avg / 2
  cmp      r1, r3  @ compare r1 to r3, i.e. a vs. avg
  bgt     Greater  @ branch to Greater if a>avg
  blt     Lesser   @ branch to Lesser if a<avg
Equal:             @ can put a label for this default,
                   @ execution proceed here if a==avg
```

Now we need to implement the default (if none of the branches are taken) as well as the bodies of the branches…

Reference C code:

```
int a, b, avg, res;
a = 100;
b = 200;
avg = (a+b)/2;
if (a>avg)  res=1;
if (a<avg)  res=-1;
if (a==avg) res=0;
```

Step 5: Handling of a==avg

```
compaavg:               @ function compaavg that compares
 …
  cmp     r1, r3   @ compare r1 to r3, i.e. a vs. avg
  bgt     Greater  @ branch to Greater if a>avg
  blt     Lesser   @ branch to Lesser if a<avg
Equal:             @ can put a label for other option,
                   @ execution proceed here if a==avg
  mov     r0, #0   @ set res = 0 and return
  b       Return   @ branches to return*
```

That’s the default… now let’s do the body of the branches

* Could simply return at this point, but I’m trying to make the code easier to follow, so am leaving just a jump to the end of this function.

Reference C code:

```c
int a, b, avg, res;
a = 100;
b = 200;
avg = (a+b)/2;
if (a>avg)  res=1;
if (a<avg)  res=-1;
if (a==avg) res=0;
```

Step 6: Handling of a>avg and a<avg branch bodies

```
compaavg:             @ function compaavg that compares
 …
Equal:                @ can put a label for other option,
                      @ execution proceed here if a==avg
  mov     r0, #0     @ set res = 0 and return
  b       Return     @ branches to return
Greater:
  mov     r0, #1     @ set res = 1 and return
  b       Return
Lesser:
  mov     r0, #0     @ -1 more complex as it is 0xFFFFFFFF
  sub     r0,r0,#1   @ instead, subtract 1 from 0 to get -1
  b       Return
```

Reference C code:

```
int a, b, avg, res;
a = 100;
b = 200;
avg = (a+b)/2;
if (a>avg)  res=1;
if (a<avg)  res=-1;
if (a==avg) res=0;
```

Step 7: Finally doing the return (if necessary)

```
compaavg:          @ function compaavg that compares
…
Lesser:
  mov     r0, #0   @ complicated since -1 = 0xFFFFFFFF
  sub     r0,r0,#1 @ instead, subtract 1 from 0 to get -1
  b       Return
Return:
  bx lr   @ standard returns from function
          @ the rule for C is that the return value is
          @ stored in r0. Which is what we have set up.
```

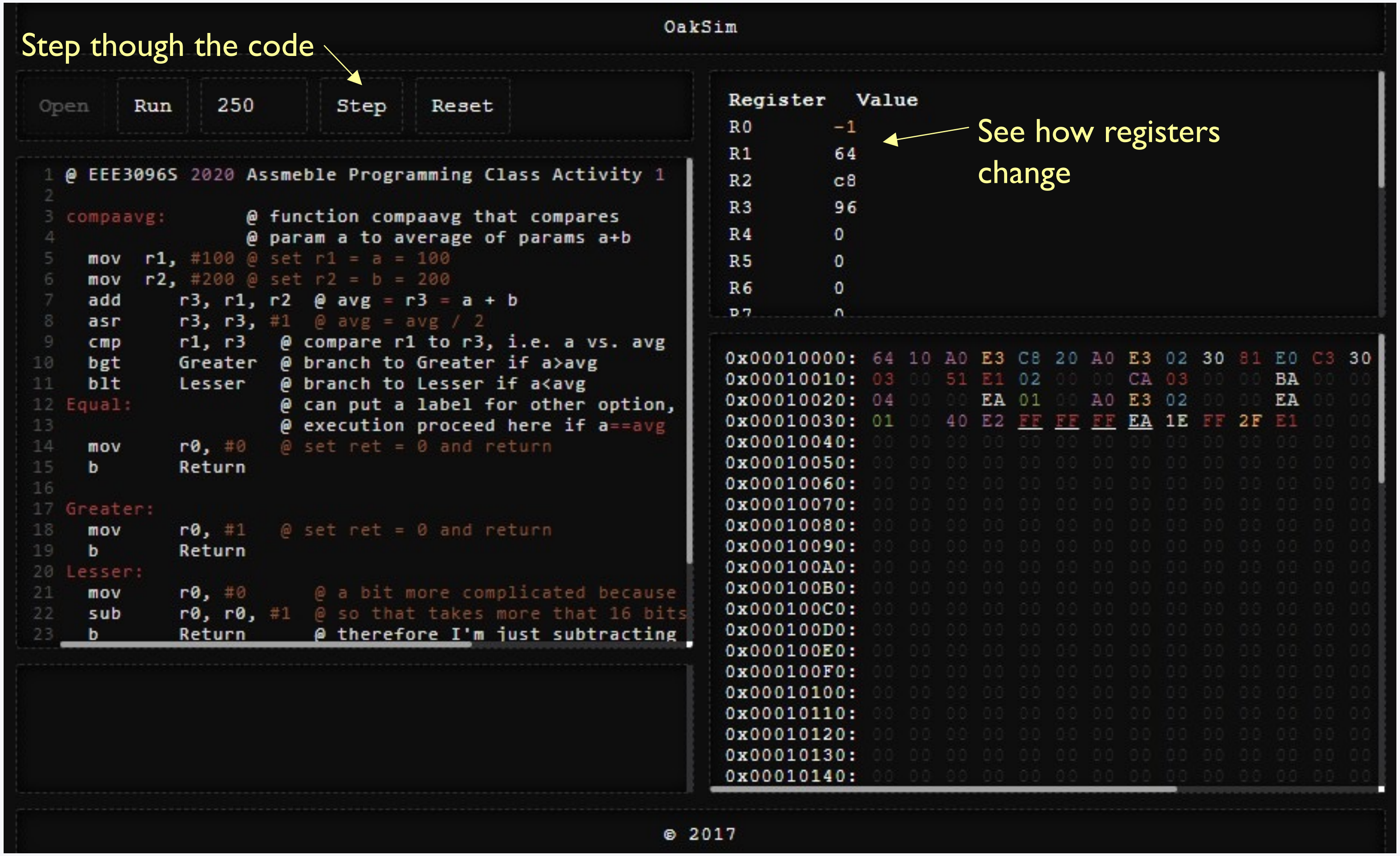
Final code:

```
compaavg:          @ function compaavg that compares
                   @ param a to average of params a+b
mov  r1, #100    @ set r1 = a = 100
mov  r2, #200    @ set r2 = b = 200
add  r3, r1, r2  @ avg = r3 = a + b
asr  r3, r3, #1  @ avg = avg / 2
cmp     r1, r3   @ compare r1 to r3, i.e. a vs. avg
bgt     Greater  @ branch to Greater if a>avg
blt     Lesser   @ branch to Lesser if a<avg
Equal:             @ can put a label for this default,
                   @ execution proceed here if a==avg
  mov     r0, #0   @ set res = 0 and return
  b       Return   @ branches to return*
Greater:
  mov     r0, #1   @ set res = 1 and return
  b       Return
Lesser:
  mov     r0, #0   @ -1 more complex as it is 0xFFFFFFFF
  sub     r0,r0,#1 @ instead, subtract 1 from 0 to get -1
  b       Return
Return:
  bx lr   @ standard returns from function
          @ the rule for C is that the return value is
          @ stored in r0. Which is what we have set up.
```

TESTING COMPAAVG.S ON OAKSIM

READ

OakSim: https://wunkolo.github.io/OakSim/
Select all the text in the left window (the Assembly code) and replace it with the compaavg code, then press Reset and step through the code to see how the registers change. You should eventually get R0 = -1.



```
@ EEE3096S 2020 Assmeble Programming Class Activity 1

compaavg:          @ function compaavg that compares
                   @ param a to average of params a+b
  mov  r1, #100 @ set r1 = a = 100
  mov  r2, #200 @ set r2 = b = 200
  add     r3, r1, r2  @ avg = r3 = a + b
  asr     r3, r3, #1  @ avg = avg / 2
  cmp     r1, r3   @ compare r1 to r3, i.e. a vs. avg
  bgt     Greater  @ branch to Greater if a>avg
  blt     Lesser   @ branch to Lesser if a<avg
Equal:             @ can put a label for other option,
                   @ execution proceed here if a==avg
  mov     r0, #0   @ set ret = 0 and return
  b       Return

Greater:
  mov     r0, #1   @ set ret = 0 and return
  b       Return
Lesser:
  mov     r0, #0      @ a bit more complicated because
  sub     r0, r0, #1  @ so that takes more that 16 bits
  b       Return      @ therefore I'm just subtracting
```

```
Register   Value
R0           -1
R1           64
R2           c8
R3           96
R4            0
R5            0
R6            0
R7            0
```

```
0x00010000: 64 10 A0 E3 C8 20 A0 E3 02 30 81 E0 C3 30
0x00010010: 03 00 51 E1 02 00 00 CA 03 00 00 BA 00 00
0x00010020: 04 00 00 EA 01 00 A0 E3 02 00 00 EA 00 00
0x00010030: 01 00 40 E2 FF FF FF EA 1E FF 2F E1 00 00
0x00010040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00010050: 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00010060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00010070: 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00010080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00010090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000100A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000100B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000100C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000100D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000100E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000100F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00010100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00010110: 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00010120: 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00010130: 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00010140: 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Assembly Activity Done

✓ Second pass of learning Assembly: writing an ARM Assembly program using real ARM instructions.

✓ Hopefully you now know the specific approach now of writing ARM Assembly that can be combined by GAS.

Of course you might get something along these lines in a test or exam, but moreover, the important aspect is knowing a bit about instructions that are commonly available on a CPU and how they work.

C & ASSEMBLY? 🤔

- This is a brief starting point to ARM coding; we have not yet looked at calling conventions for Assembly routines, the passing of parameters, using the stack, etc.
- Next lecture will clarify these issues and how to mix C **and** Assembly code; nowadays it is rare that you would ever write a program entirely in Assembly :)

RECOMMENDED LEARNING RESOURCES

- The following YouTube video covers the essentials of Assembly programming:

https://www.youtube.com/watch?v=FV6P5eRmMh8