# SENG440 Embedded Systems

– Lesson 100: Project Pilot Lesson–

## Mihai SIMA

**msima@ece.uvic.ca**

Academic Course

## Copyright © 2018 Mihai SIMA

## Disclaimer

The purpose of this course is to present general techniques and concepts for the analysis, design, and utilization of embedded systems. The requirements of any real embedded system can be intimately connected with the environment in which the embedded system is deployed. The presented design examples should not be used as the full design for any real embedded system.

## Lesson 100: Project Pilot Lesson

1 Introduction

2 List of Projects

3 Report organization

4 Project grading scheme

5 Project goals

6 Project scenario

7 Application-specific ARM instructions

8 Firmware approach

9 Hardware solution

10 Project design flow – review

11 Understanding the algorithm

12 The complexity of the design task

13 Optimization Flags

## Introduction

- Example: implement Discrete Cosine Transform on an:
    - ARM-based system in which
    - the **butterfly** operation benefits from hardware/firmware support
- Goal: to practice the navigation in the hardware-software space
- What you practice for project is also good for mid-term and final
    - Fixed-point arithmetic
    - Software optimization techniques
- The text will be typed in 11- or 12-point type, single spaced
- The references will be listed and numbered at the end of the report
- The report will contain no more than 20 pages

## List of Projects

- Color Space Conversion (RGB-to-YCC and YCC-to-RGB)
- Audio compression and decompression (A law, $\mu$ law)
- Discrete Cosine Transform and Inverse Discrete Cosine Transform
- RSA cryptography
- Matrix inversion
- Matrix diagonalization (SVD, EVD)
- FIR filtering
- Hufmann encoding and decoding
- Motion estimation
- COordinate Rotation DIgital Computer (CORDIC)
- Convergence Computing Method (CCM)

## Report organization

- **Front page** with title, author, affiliation and contact information (UVic, Dept. of CS/ECE, student number, e-mail) and a blank dotted area to be filled in with the submission date at the time of submission
- **Introduction** including a short description of the domain, performance requirements, an enumeration of the contributions, and project organization.
- **Theoretical background**
- One or more sections regarding **the design process** itself
- **Performance/cost evaluation**
- **Conclusions**
- **Bibliography**

## Project grading scheme

| Task | Points |
|------|-------:|
| Design requirements and specifications (UML) | 10 |
| Functionally-correct C code | 5 |
| Optimized C code | 10 |
| Optimized assembly code | 10 |
| Firmware | 5 |
| Hardware (automaton design, timing estimation) | 5 |
| **Final report (mandatory)** | 15 |
| Interview (points per failure) | (-2) |
| **Total** | **60** |

- If your score is $\geq 50$, you will get 50 points.

## Project goals

- To expose students to a complete hardware-software co-design flow:
  1. Specification and modelling (define the specs, UML charts, blue-prints)
  2. Pure-software implementation (C and assembly code)
  3. *Profile* the resulting executable and detect the bottleneck
  4. Augment the standard instruction set with a new instruction
  5. Estimate the latency of this new instruction by:
     - Designing the computing unit that implements the newly defined operation
     - *Educated guess*
  6. Instantiate the new instruction (intrinsic or assembly inlining)
  7. Build a testbench and test the system
  8. Determine the performance improvement
  9. Disseminate the information (technical report)

- Resulting engine: **Application-Specific Instruction-set Processor (ASIP)**

## Project scenario – example

Implement the $8 \times 8$ DCT on an ARM-based system in which the butterfly operation will benefit from hardware/firmware support

1. You will describe the algorithm in a high-level language (C). The operation that will benefit from hardware/firmware support will be implemented first as a routine, then as a *inline* routine, and finally as a macro.

2. You will debug the high-level implementation by compiling it on a workstation. Profile-driven compilation may be needed. You will also build the testbench (a set of input data with known output data).

3. You will generate ARM assembly code for $8 \times 8$ DCT and *estimate* the performance of the pure-software solution. In particular, you will determine the performance for all three butterfly software implementations: routine, *inline* routine, and macro.

Mihai SIMA © 2018 Mihai SIMA

## Project scenario – example (cont'd)

4. You will use software optimization techniques (such as software pipelining or loop unrolling) to improve the performance.

5. You will optimize by hand the assembly code. You will assemble the code and run the executable on the ARM-based system available on intranet.

6. Instead of calling or inlining the butterfly routine, you will instantiate a butterfly operation. You will compile the code (note that you will not be able to assemble it.) To determine the peformance (that is, the latency) of the butterfly operation, you will implement it in hardware and firmware.

7. You will implement the butterfly in firmware. As microcoded engine you will use the ARM instruction set without multiplication and division. You will propose two solutions for a 1-issue slot, and a 2-issue slot microcoded engine, respectively. You will determine the computing performance for both solutions (that is, you will count the number of cycles by hand, since you do not have access to a simulator for the microcoded engine).

## Project scenario – example (cont'd)

8 You will implement the butterfly in hardware. An automaton with either D or J-K flip-flops will be designed. You will also create the testbench (which will be also written in VHDL). To determine the butterfly latency, you will compile the VHDL code and simulate the resulting circuit.

9 You will describe the computing scenario in UML and build the blue-prints of the design.

10 You will send me by e-mail the C, assembly, VHDL, and UML files.

11 You will write the report and submit it in person.

12 You will submit the report (hardcopy) in person and answer questions.

Mihai SIMA                                                                                                © 2018 Mihai SIMA

## ARM augmented with application-specific instructions

- We want to implement an algorithm on an ARM
- Let us assume that software solution is not fast enough
- Amdahl rule: 10% code takes 90% computing time
- We profile the code and figure out the bottleneck
- Is it worth addressing the bottleneck? Very likely, due to Amdahl rule
- The idea is to design an application-specific computing unit, and augment the instruction set with a new instruction that calls this unit
- The new computing unit can be either
    - implemented in firmware
    - implemented in customized hardware
- If the firmware solution gives us good performance, we usually stop here

## Architectural constraints in augmenting ARM

- There is an upper limit for the number of input and output operands (2 inputs and 1 output for ARM)
- More operands means we have to build a <u>non-reentrant</u> unit, and use this unit as follows: Read, Read, .., Read, Execute, Write, ..., Write.
- A non-reentrant unit rises another set of problems, for example:
  - Should the interrupts be disabled during the non-reentrant unit is active?
  - What is happening if an exception is encountered while the non-reentrant unit is active? What elements we need to provide in order to restore the state of the non-reentrant unit after returning from exception
- The latency of most units ranges from 1 to 3 cycles – is it possible to define a unit with a latency arbitrary large?
- Obviously, we can design a pipeline unit, but this rises compilation problems – optimization when one of the units is 'non-standard' is difficult

Mihai SIMA © 2018 Mihai SIMA

# Firmware approach (third-order systems or higher)

- Our application-specific instruction is executed as a microprogram
- Control Store is a 'non-standard' memory
- In fact, the microinstruction set is 'non-standard'
- Vertical microcode – a single (micro)-computing unit is controlled per cycle (1 issue-slot)
- Horizontal microcode – multiple (micro)-computing units are controlled per cycle (2 issue-slots)

SENG440 Embedded Systems (100: Project Pilot Lesson)
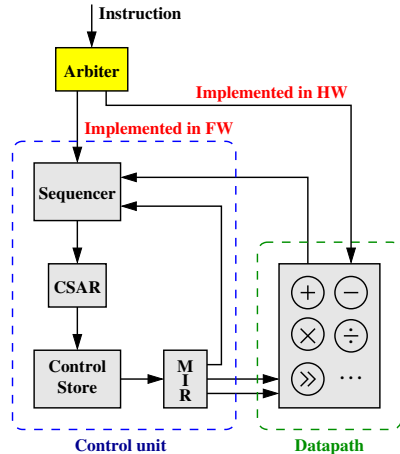
## Vertical and horizontal microcode

- Horizontal microcode: the constituents of the microcode engine are replicated in order to execute multiple micro-instructions
- 2 issue-slots horizontal microcode: think of this engine as having:
  - Double-wide control store, each half storing a vertical microinstructions
  - Dual-port register file, such that both microinstructions can simultaneously get operands and write back results
  - Complex sequencer (condition codes can come from both microinstructions)
- The price for a 2 issue-slot microcode engine is larger, much larger than twice the price of a vertical microcode engine
- The largest performance improvement that can be achieved: $2\times$
- A lot of issue-slots will contain NOP, since there are dependencies that make difficult to find operations to fill in all the issue-slots
- Is it worth the price?

Mihai SIMA                                                                                                    © 2018 Mihai SIMA

## Vertical microcode – an example

- Let us assume a processor whose instruction set includes a 32-bit bit-wise AND instruction

- For our example, we assume that the 32-bit bit-wise AND is a 'complex' instruction which is executed in microcode

- Let us consider an implementation that performs the AND operation serially 1-bit per cycle
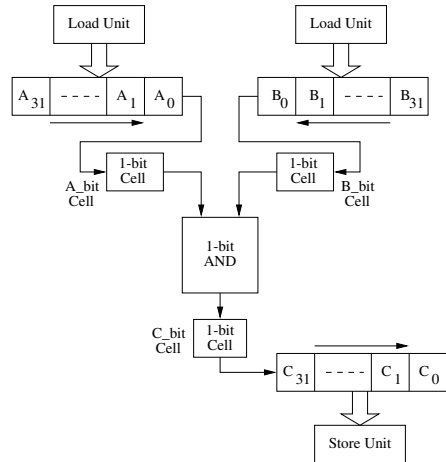
## Vertical microcode – an example (cont'd)

```
01        | LOAD  A                |
02        | LOAD  B                |
03  loop: | SHIFT A                |
04        | SHIFT B                |
05        | AND   A_bit,B_bit,C_bit |
06        | SHIFT C                |
07        | BACK  loop, 31_TIMES   |
08        | STORE C                |
09        | ENDOP                  |
```

- 9 cycles are needed
- There are not so many dependencies, thus horizontal microcode may provide for significant speed-up

Mihai SIMA © 2018 Mihai SIMA

## Horizontal microcode – an example

```
01          | LOAD  A                 | LOAD  B              |
02   loop:  | SHIFT A                 | SHIFT B              |
03          | AND   A_bit,B_bit,C_bit | NOP                  |
04          | SHIFT C                 | BACK loop, 31_TIMES  |
05          | STORE C                 | ENDOP                |
```

- 5 cycles are needed
- Speed-up of almost $2\times$

Mihai SIMA                                                                                      © 2018 Mihai SIMA

## What to do for the 'firmware' section of project

- Write / generate assembly code for the bottleneck function

- Assume a 2 issue-slot microcode engine and try to fill in with operations as many issue-slots as possible. This is the horizontal microcode

- Compare the 1 issue-slot solution with the 2 issue-slot solution

- How close are you to the maximum speed-up of $2\times$?

- Compare the 1 issue-slot solution with the 3 issue-slot solution

- How close are you to the maximum speed-up of $3\times$?

## Hardware solution: full-custom computing unit

- Let us assume that the firmware solution is still not fast enough
- We have to implement the computing unit in hardware, that is, we have to design a new circuit
- Digital design, VHDL, mapping to silicon, etc.
- We are trying to determine performance (latency) and cost (for example, the number of logic gates, or silicon area)
- Latency – we need to have:
    - The physical model of ARM processor
    - Software tools to do mapping, back-annotation, simulation
- Estimation is still possible although we do not have these resources
- **Addition takes 1 cycle to complete**

## Hardware solution

- Zeroth-order system: combinational circuit
  - From register file back to register file in <u>one step</u>
  - It is not possible to issue a new instruction before the current instruction completes its processing

- First-order system: pipeline
  - From register file back to register file in <u>multiple steps</u>
  - It is now possible to issue a new instruction before the current instruction completes (in this case we have a pipelined processor)
  - Appropriate when large amount data is processed in a repetitive fashion

- Second-order system: automaton
  - Complex instructions can be implemented

## ARM simulator – a simple C program

```c
#include <....h>

int main( int argc, char *argv[]) {
  int i, j, k=0;

  if ( argc != 2) {
    printf( "Enter exactly one argument\n");
    exit( 1); /* shell errors... */
  }

  i = atoi( argv[1]);
  for( j=0; j<i; j++)
    k += j;
  printf( "k = %i\n", k);
  exit( 0);
}
```

Mihai SIMA © 2018 Mihai SIMA

## ARM simulator (cont'd)

- You can *ssh* into any of the Lab machines (ugls.ece.uvic.ca) with your Netlink credentials
- Compile with static libraries, since the simulator might not support dynamic libraries yet
  **arm-none-linux-gnueabi-gcc -static -march=armv5 file.c -o file.exe**
- Simulate the resulting executable (file) with:
  **qemu-arm file.exe**
- The output of the simulator

  k = . . .

- Alternatively, you can use the physical ARM computer

Mihai SIMA        © 2018 Mihai SIMA

## The project design flow – review

- We have to implement algorithm **A**
- We write high-level (C) code to implement algorithm **A**, we then compile, simulate, profile. Assume that we find out that function **B** is the bottleneck
- We now try to optimize function **B**:
    - We first try to inline the function **B**
    - We then optimize the assembly code we get by compilation
    - We finally write function **B** in assembly
- Assume that, even after all optimizations, function **B** is still the bottleneck
- We now define a new instruction **EXECUTE_B** and design a new computing unit implementing **B**:
    - Firmware solution: the new computing unit is microcoded
    - Hardware solution: custom hardware is designed for a new computing unit

## Instantiating machine-level custom instructions

- Assume we have the following C code:

```
int my_function( int a, int b) {
  <do something on a and b>
  return ...
}

int main( void) {
  int m, n, s;
  s = my_function( m, n);
}
```

- We want to implement my_function function in hardware/firmware
- That is, we build new hardware/firmware and **extend** the machine instruction set with a new instruction: **my_function**

Mihai SIMA                                                                                                              © 2018 Mihai SIMA

SENG440 Embedded Systems (100: Project Pilot Lesson)

## Instantiating machine-level custom instructions (cont'd)

- How to call the newly defined machine instruction from within C?
- By inlining assembly code:

```
int m, n, s;
__asm__ ( "my_function %0, %1, %2" : "=r" (s) : "r" (m), "r"
```

- Compile: **arm-none-linux-gnueabi-gcc -static -march=armv5 -S file.c**
- The resulting assembly code:

```
my_function R1, R2, R3
```

- Do not expect to be able to assemble such code, since the assembler cannot allocate an operation code for **my_function**
- Augmenting the assembler with new opcodes is beyond the project scope

Mihai SIMA                                                                    © 2018 Mihai SIMA

SENG440 Embedded Systems (100: Project Pilot Lesson)

## A simple program for addition

```
#include <stdio.h>
volatile int c;

int add( int a, int b) {
  return a + b;
}

int main( void) {
  int a = 1, b = 2;
  c = add( a, b);
  printf( "a + b = %i\n", c);
  exit( 1);
}
```

To compile just type:
**arm-none-linux-gnueabi-gcc -static -march=armv5 addition.c** <enter>

Mihai SIMA                                                                                    © 2018 Mihai SIMA

## A simple program for addition

```
add:
     mov      ip, sp
     stmfd    sp!, {fp, ip, lr, pc}
     sub      fp, ip, #4
     sub      sp, sp, #8
     str      r0, [fp, #-16]
     str      r1, [fp, #-20]
     ldr      r3, [fp, #-16]
     ldr      r2, [fp, #-20]
     add      r3, r3, r2
     mov      r0, r3
     b        .L2
.L2:
     ldmea    fp, {fp, sp, pc}
```

To assemble just type:
**arm-none-linux-gnueabi-gcc
-static -march=armv5 addition.s
-o addition** <enter>

To simulate just type:
**qemu-arm addition**

## How important is to understand the algorithm

- The solution of the following system is $x = 1$ and $y = -1$

$$835x + 667y = 168$$
$$333x + 266y = 67$$

- The solution of the following system is $x = -666$ and $y = 834$

$$835x + 667y = 168$$
$$333x + 266y = 66$$

- The solution changes dramatically when the right term 67 becomes 66
- Recall from Mathematics:

> *A system of linear equations is said to be ill-conditioned when*
> *some small perturbation in the system can produce large*
> *changes in the exact solution*

## How important is to understand the algorithm

- The other way around: check if $x = -666$ and $y = 834$ is the solution of the first system:

$$835 \cdot (-666) + 667 \cdot 834 - 168 = 0$$
$$333 \cdot (-666) + 266 \cdot 834 - 67 = -1$$

- The last solution exactly satisfies the first equation and comes very close to satisfying the second
- Naive belief: the solution $(666, 834)$ is very close to the exact one
- **It is not true that you can always check the accuracy of the computation by substituting it back into the original equations**
- This also raises the problem of number representation and precision
  - How many bits to represent the coefficients are needed?

# The complexity of the design task

- **Algorithm design**
  - Knowledge of the particular domain you are working on
  - Mathematics
  - Computer arithmetic

- **Software design**
  - Programming languages
  - Optimization techniques
  - Computer architecture and implementation

- **Hardware design**
  - Digital design
  - Computer arithmetic
  - Hardware description languages

## Optimization Flags – Example

```c
#include <stdio.h>


volatile int a, b, c, d, e;
int f_01( int a, int b,
          int c, int d) {
  register int x, y;
  x = 0;
  if( c > 0)
    x = a + (b << 1);
  if( d < 0)
    y = (a >> 1) + b;
  else
    y = (a >> 1) - b;
  return x*y;
}
```

```c
int main( void) {
  printf( "a = ");
  scanf( "%i", &a);
  printf( "b = ");
  scanf( "%i", &b);
  printf( "c = ");
  scanf( "%i", &c);
  printf( "d = ");
  scanf( "%i", &d);

  e = f_01( a, b, c, d);

  printf( "e = %i\n\n", e);
  return( 0);
}
```

## arm-linux-gcc -S test.c

```
f_01:    str      fp, [sp, #-4]!          .L2:    ldr      r3, [fp, #-20]
         add      fp, sp, #0                      cmp      r3, #0
         sub      sp, sp, #28                     bge      .L3
         str      r0, [fp, #-8]                   ldr      r3, [fp, #-8]
         str      r1, [fp, #-12]                  mov      r2, r3, asr #1
         str      r2, [fp, #-16]                  ldr      r3, [fp, #-12]
         str      r3, [fp, #-20]                  add      r3, r2, r3
         mov      r2, #0                          str      r3, [fp, #-24]
         str      r2, [fp, #-28]                  b        .L4
         ldr      r3, [fp, #-16]          .L3:    ldr      r3, [fp, #-8]
         cmp      r3, #0                          mov      r2, r3, asr #1
         ble      .L2                             ldr      r3, [fp, #-12]
         ldr      r3, [fp, #-12]                  rsb      r3, r3, r2
         mov      r2, r3, asl #1                  str      r3, [fp, #-24]
         ldr      r3, [fp, #-8]          .L4:    ldr      r3, [fp, #-28]
         add      r3, r2, r3                      ldr      r2, [fp, #-24]
         str      r3, [fp, #-28]                  ...
```

## arm-linux-gcc -marm -O1 -S test.c

```
f_01:
        mov     ip, r3
        cmp     r2, #0
        movle   r3, #0
        movgt   r3, r1, asl #1
        addgt   r3, r3, r0
        cmp     ip, #0
        addlt   r1, r1, r0, asr #1
        rsbge   r1, r1, r0, asr #1
        mul     r0, r3, r1
        bx      lr
```

# arm-linux-gcc -mthumb -O1 -S test.c

```
f_01:
        push    {r4, lr}
        mov     r4, r3
        cmp     r2, #0
        bgt     .L2
        mov     r2, #0
        b       .L3
.L2:
        lsl     r3, r1, #1
        add     r2, r3, r0
.L3:
        cmp     r4, #0
```

```
        bge     .L4
        asr     r3, r0, #1
        add     r1, r3, r1
        b       .L5
.L4:
        asr     r3, r0, #1
        sub     r1, r3, r1
.L5:
        mov     r0, r2
        mul     r0, r1
        pop     {r4}
        pop     {r1}
        bx      r1
```

## arm-linux-gcc -marm -Os -S test.c

```
f_01:
        cmp     r2, #0
        addgt   r2, r0, r1, asl #1
        movle   r2, #0
        mov     r0, r0, asr #1
        cmp     r3, #0
        addlt   r1, r0, r1
        rsbge   r1, r1, r0
        mul     r0, r2, r1
        bx      lr
```

# arm-linux-gcc -mthumb -Os -S test.c

```
f_01:
        push    {r4, lr}                        cmp     r4, #0
        mov     r4, r3                          bge     .L4
        cmp     r2, #0                          add     r1, r0, r1
        bgt     .L2                             b       .L5
        mov     r3, #0                  .L4:
        b       .L3                             sub     r1, r0, r1
.L2:                                    .L5:
        lsl     r3, r1, #1                      mov     r0, r3
        add     r3, r3, r0                      mul     r0, r1
.L3:                                            pop     {r4}
        asr     r0, r0, #1                      pop     {r1}
                                                bx      r1
```

# Questions, feedback