Lesson 100: The project

- 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 (and vice-versa)
  - Fixed-point arithmetic
  - Software optimization techniques
  - SpecCharts language
- 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 available projects

- Color Space Conversion (RGB-to-YCC and YCC-to-RGB)
- Audio compression and decompression (A law, μ law)
- Discrete Cosine Transform and Inverse Discrete Cosine Transform
- Bit scrambling
- Matrix inversion
- Matrix diagonalization
- FIR filtering
- Hufmann encoding and decoding
- Motion estimation
- COordinate Rotation DIgital Computer (CORDIC)

Report organization

1. **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
2. **Introduction** including a short description of the domain, performance requirements, an enumeration of the contributions, and project organization.
3. **Theoretical background**
4. One or more sections regarding **the design process** itself
5. **Performance/cost evaluation**
6. **Conclusions**
7. **Bibliography**

Grading guideline

Task	Points
Functionally-correct C code	5
Optimized C code	10
Optimized assembly code	5
Firmware	5
Hardware (VHDL behavioral, timing estimation)	5
Hardware (automaton design, SpecCharts)	5
UML description	5
Final report	15
Interview (points per failure)	(-2)
Total	**55**

- If your score is ≥ 50, you will get 50 points.

Project goals

- To expose the student to a complete hardware-software co-design flow:
  1. Specification and modelling (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. Design the computing unit that implements the newly defined operation
     - Zeroth-order system: combinational circuit (VHDL)
     - First-order system: pipeline (VHDL)
     - Second-order system: automaton (VHDL, SpecCharts)
     - Third-order system: firmware
  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)
- The resulting engine: **Application-Specific Instruction-set Processor (ASIP)**

Project scenario

Example:

Implement the 8×8 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 (preferably 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 and output data), which becomes part of your project.
3. You will generate ARM assembly code for 8×8 DCT and estimate the performance of the pure-software solution by simulation on a cycle-accurate simulator. In particular, you will determine the performance for all three butterfly software implementations: routine, *inline* routine, and macro.

Project scenario (cont'd)

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

5. You will optimize by hand the resulting assembly code. You will assemble the code and simulate the resulting executable on the cycle-accurate simulator.

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 (cont'd)

8. You will implement the butterfly in hardware. An automaton with either D or J-K flip-flops will be designed. Automaton description in either SpecCharts or VHDL is required. 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, SpecCharts/VHDL, and UML files.
11. You will write the report and submit it in person.
12. You will answer a couple of questions at submission time.

ARM augmented with application-specific instructions

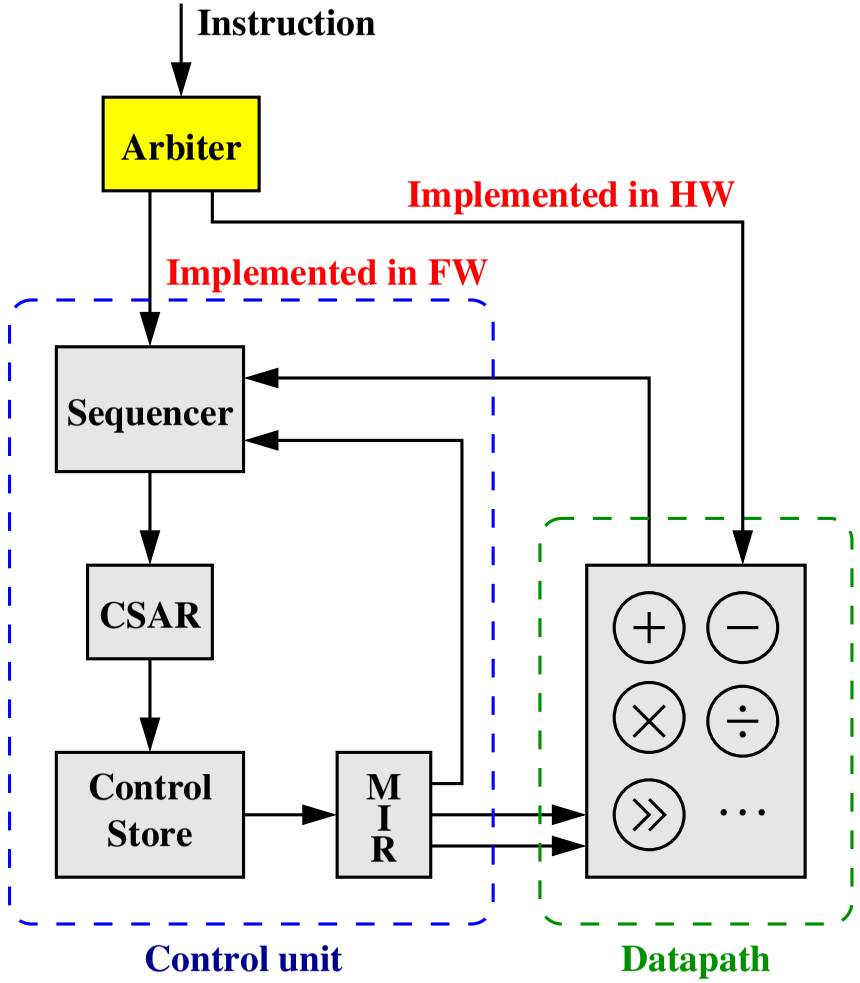
- We want to implement an algorithm on an ARM
- Let us assume that software solution is not good enough (too slow)
- Amdahl rule: 10% code takes 90% computing time
- We profile the code and figure out the bottleneck
- Is it worth addressing the bottleneck? Very likely, due to Amdahl rule
- The idea is to design an application-specific computing unit, and augment the instruction set with a new instruction that can call the computing 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 *non-reentrant* 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

Firmware solution (third-order system or more)

- 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)

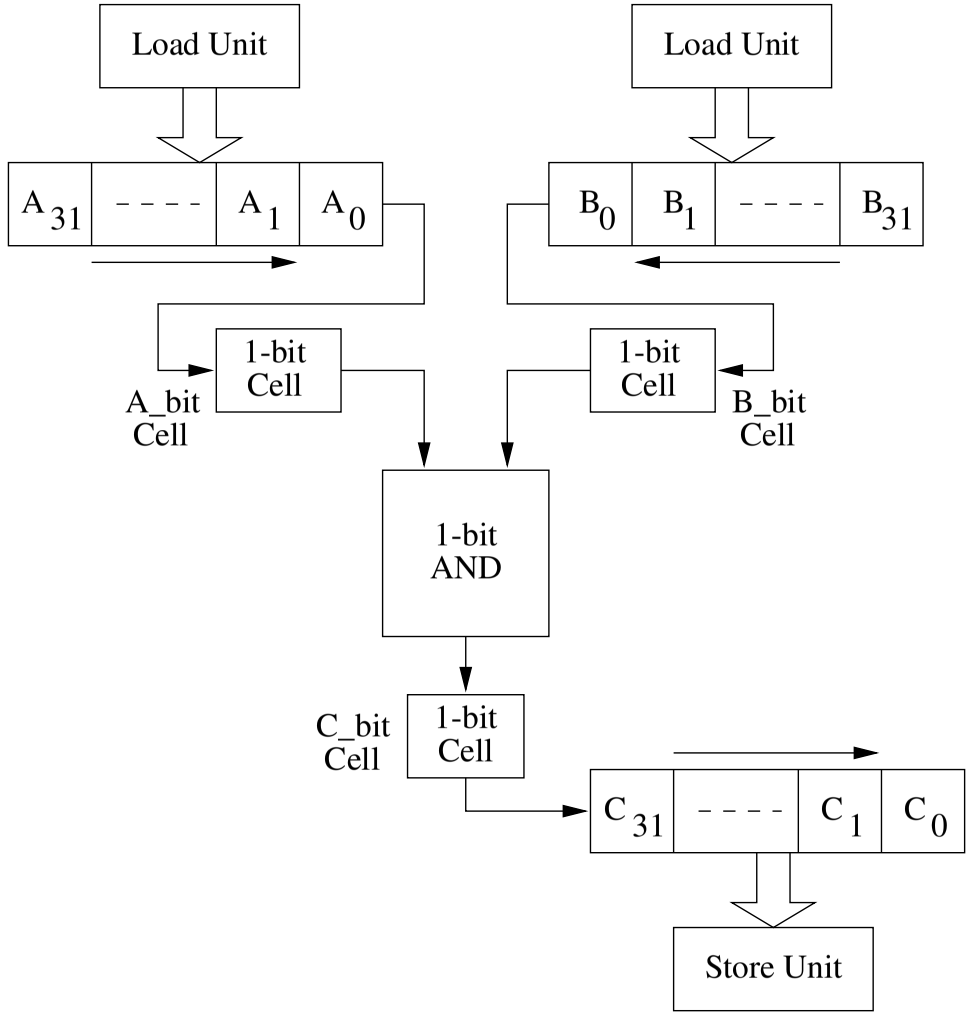


Vertical and horizontal microcode

- Horizontal microcode: the constituents of the microcode engine are duplicated in order to execute multiple micro-instructions
- Our case: 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
  - More complex sequencer, since 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 theoretically achieved: 2×
- 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?

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

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×

What to do for the 'firmware' section of project

- Write assembly code for the bottleneck function
- Please note that while writing assembly code (which is typically tightly optimized) does not have effect on the overhead for calling routines (save registers, stack pointer, ...)
- Defining a new instruction and executing it as microcode, removes this overhead
- As a microinstruction set use a simplified version of ARM instruction set
- Rewrite assembly without using 'complex' instructions, like multiplication, division, predicate instructions, etc. This is the *vertical microcode*
- 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

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 *one step*
  - It is not possible to issue a new instruction before the current instruction completes
- First-order system: pipeline
  - From register file back to register file in *multiple steps*
  - It is now possible to issue a new instruction before the current instruction completes (in this case we have a pipelined processor)
  - Appropriate when a large amount data is to be processed in a repetitive fashion
- Second-order system: automaton
  - Complex instructions can be implemented

ARM simulator – assume a simple C program

```
#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);
}
```

ARM simulator (cont'd)

- You can *ssh* into any of the Lab machines (c66.seng.uvic.ca to c101.seng.uvic.ca) with your Netlink credentials

- Compile with static libraries, since the simulator does not support dynamic libraries yet

  arm-linux-gnueabi-gcc -static -march=armv5 file.c -o file.exe

- Simulate the resulting executable (file) with sima (simulator ARM)

  qemu-arm file.exe

- The output of the simulator

  ```
  k = ...
  ```

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 have 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 the 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 the 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**

Instantiating machine-level custom instructions (cont'd)

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

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

- Compile: **arm-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 instructions is beyond the project scope

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-linux-gnueabi-gcc -static -march=armv5 addition.c**
<enter>

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-linux-gnueabi-gcc -static -march=armv5 addition.s -o addition <enter>

To simulate just type:
qemu-arm addition

The importance of Mathematics

- 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

The importance of Mathematics

- 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 simply 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 our 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

Questions, feedbacks