10 – DSP Firmware

Oscar Gustafsson

Todays lecture

- DSP firmware
- Application modelling
- Hardware verification

On to firmware development issues
A case study on MP3 decoding

- The MPEG1 Layer III specification gives the procedure for MP3 decoding but does not say exactly how the calculations should be performed
- A decoder may use
  - Floating-point or fixed-point (or more esoteric number representations…)
  - Different algorithms for the various filters

Case study: Modelling MP3 decoding

- A compliant MP3 decoder will decode a certain test bitstream without deviating too much from a reference output in the standard
  - A fully compliant MP3 decoder has an RMS error of less than $2^{-15}/\sqrt{12}$ and an absolute difference of less than 2^{-14} relative to full scale
  - A limited accuracy MP3 decoder has an RMS error of less than $2^{-11}/\sqrt{12}$

Root Mean Square (RMS)

- $D_{\text{RMS}} = \sqrt{((R_1 - r_1)^2 + (R_2 - r_2)^2 + \ldots + (R_N - r_N)^2)/N}$
- For an MP3 decoder the root mean square error should be less than either $2^{-15}/\sqrt{12}$ (or $2^{-11}/\sqrt{12}$ for a limited accuracy decoder)

Absolute error

- $D_{\text{ABSMAX}} = \max\{|R_1 - r_1|, |R_2 - r_2|, \ldots, |R_n - r_n|\}$
- For an MP3 decoder the absolute error should be less than 2^{-14} for a fully compliant decoder

Signal to noise ratio (SNR)

- $\mathrm{SNR} = 20\log_{10}(\max_{\mathrm{headroom}} / D_{\mathrm{RMS}})$ dBV
- Signal to noise ratio is not used for MP3 decoding compliancy but is often used in other DSP systems

Case study: Modelling MP3 decoding

- Download MP3 decoder source code
- Instrument source code with custom functions for fixed or floating point arithmetic

```
struct NUMBER {
  int32_t exponent;
  int32_t mantissa;
};
// Floating point add
void add(struct NUMBER *result,
         struct NUMBER *x,
         struct NUMBER *y);
```

Case study: Modelling MP3 decoding

```
// Without instrumentation
float even[size/2];
float odd[size/2];
for(i=0; i<size/2; i++) {
  even[i] = in[i] +
            in[size-1-i];
}
```

```
// With instrumentation
NUMBER even[size/2];
NUMBER odd[size/2];
for(i=0; i<size/2; i++) {
  add(&even[i],&in[i],
      &in[size-1-i]);
}
```

- You can (and probably should) use operator overloading in C++ here

Case study: Modelling MP3 decoding

- Replace inefficient algorithms with faster algorithms
  - Matrix multiplication based DCT: 2048 MUL, 2048 ADD
  - Fast DCT: 80 MUL, 209 ADD

Case study: Modelling MP3 decoding

- Analyze needed mathematical operations
  - $+,\ -,\ \times,\ /$
  - $x^{4/3}$
  - $\sin, \cos, \tan$

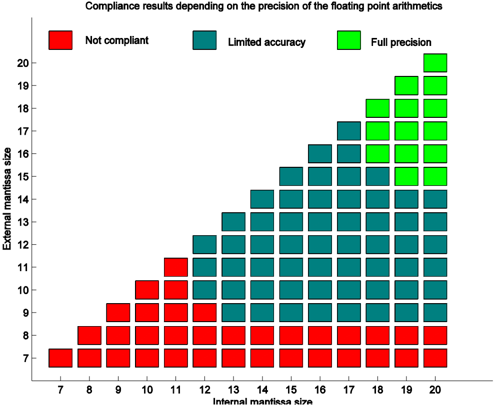
Case study: Modelling MP3 decoding

- $+,\ -,\ \times,\ /$
  - Division by constant $\Rightarrow$ Multiply with 1/constant
  - Division by power of 2: Shift (or multiply with 1/constant)
- $x^{4/3}$
  - Too large for lookup table (Number range: 0–8207)
  - Newton-Raphson on $x^{-1/3}$ requires only $+$, $-$, and $\times$
- $\sin,\ \cos,\ \tan$
  - Only used on constant values $\Rightarrow$ Can be precalculated and put in a relatively small lookup table

Case study: Modelling MP3 decoding

- Final task: rewrite reference decoder to use only $+$, $-$, and $\times$
- Also add two number formats
  - One for floating-point format in memory
    - `struct NUMBER`
  - One for floating-point format in registers
    - `struct REGISTER`

Case study: Modelling MP3 decoding



- We needed 5 exponent bits in the memory to get the required dynamic range and 6 exponent bits in the registers

Case study: Modelling MP3 decoding

- We also needed to verify that the theoretical results have a grounding in reality using listening tests
- ABX listening tests
  - The test subject gets three audio files:
  - A: Reference result
  - B: Our result
  - X: User should decide whether this file is A or B
  - (Double blind test, no human knows whether A or B is the reference file until after the fact.)

Summary of MP3 decoding

- Result of compliance test according to the standard
  - We required 9 bits of mantissa in memory and 12 bits of mantissa in registers for limited accuracy
- Result of our ABX listening test
  - We needed 10 bits of mantissa in memory and 16 bits of mantissa in registers to get high quality decoding

Summary of MP3 decoding

- The use of fast DCT algorithms had little impact on the RMS
- Using only $+$, $-$, and $\times$ was not a problem for this application except for $x^{4/3}$ which could be solved with Newton-Raphson for $x^{-1/3}$
  - Although this was changed to a polynomial approximation later on

Conclusions: Application modelling

- We need something to test
  - Instrument reference application
  - Write new application using matlab, C, etc
- We need some way to evaluate our results
  - RMS, SNR, Absolute error
    - Based on the standard or other requirements
  - Subjective tests (ABX and other double blind tests)

Conclusions: Application modelling

- We need to use reasonable datatypes
  - Fixed-point with appropriate bit widths
  - Floating-point with appropriate bit widths
- We need to use reasonable algorithms
  - FFT, Fast DCT, Newton-Raphson, CORDIC, lookup tables, etc...
  - Algorithms need to be adaptable to our HW

Conclusions: Application modelling

- Finally, our algorithms must use reasonable sized program, data and constant memory
  - We do not want megabytes of lookup-tables

Other issues

- On-chip/off-chip memory usage?
  - DMA?
  - Cache?
  - Memory organization? (e.g. tile-based or linear? Multibank?)
- Interrupt latencies
  - Reserve registers for interrupts? (In software or hardware?)

DSP Firmware

- Challenges compared to normal desktop applications
  - Real-time requirements
  - Low memory requirements
  - Specialized processors with limited compiler support
  - Often cumbersome to fix bugs using software updates

Writing large assembly programs

- Avoid this.
  - But if you do not have a compiler you do not have a choice
- You need a reference code in a high level language
  - You get to play C to assembly compiler yourself
  - At every step you should be able to compare the intermediate output from your C code with your assembler output

C-code and other high level languages

- The closer the C-code to HW, the better can be the result from the C-compiler
- Understand the compiler in detail
  - gcc -S
- Annotate enough ”Compiler known” functions
- Functional verification of compiled code
  - Do not forget the regression suite for SW!

C-code and other high level languages

- Inline assembler
  - Use C for everything but the most critical loops
  - Use inline assembler for these
- Do not optimize before you benchmark!

C-code and other high level languages

- Try to save memory
    - Know your C compiler output
    - It may be a good idea to allocate memory statically
    - Do not use dynamic memory allocation if you can avoid it (`new`, `malloc`)
    - Do not use huge library functions out of convenience (`printf` vs `puts`)
    - Do not use floating point math if your DSP processor does not have HW support for it…

Low cycle cost assembly kernels

- Gain much by saving cycles in an inner loop!
- Use `REPEAT` instead of conditional jump
- Loop unrolling
- The code cost of inner loops is not so important!
- Use as many vector instruction as possible
- Keep useful data in RF as long as possible
- Use conditional execution if needed
    - Exception: Modern OoO processors with good branch predictors

When to benchmark/profile?

- When the project has reached:
  - Pen and paper
    - Can this be done for lab 4? Try it!
  - Application modeling
    - Crude MIPS count available here based on instrumentation for example
  - ASIP instruction selection
  - Firmware development

Tools of the trade

- For calculating $\sin / \cos / \tan$, $x^{4/3}$, etc there are many possibilities
  - CORDIC
  - Lookup tables
  - Newton-Raphson
  - Polynomial
  - etc
- Combinations are also possible
  - First lookup-table, then a few Newton-Raphson iterations

Tools of the trade

- Algorithmic strength reduction
  - FFT/DCT/etc
  - Fast Fir Algorithms (FFA)
  - Fast Matrix $\times$ Vector multiplications (Strassen)
  - …

Tools of the trade

- Fast Matrix multiplication
  - Winograd's inner product
    - (Can be used for FIR filters as well to halve the number of multiplications (ISCAS2013))
  - Strassen ($O(2^{2.8})$) `http://en.wikipedia.org/wiki/Strassen_algorithm`
  - (Later work brings down the complexity to less than $O(2^{2.4})$, but only for very large matrices.)

Tools of the trade

- Strength reduction algorithms often show better performance on paper than in real benchmarks
  - Reason: caches, branch prediction, addressing irregularities, etc
- However, we are free to design a processor in whatever way necessary $\Rightarrow$ such algorithms may make more sense for ASIPs than general purpose processors.

Tools of the trade (Esoteric)

- Esoteric way of calculating 1/sqrt(x) where x is a floating point value
  - Google 0x5f3759d5
- Fun reading: Hacker's delight
  - Tips for various bit-level manipulations, etc
  - Example: Why would you calculate `x & (x-1)`?
  - Or `x & (-x)`?
- See also `http://aggregate.org/MAGIC/` and `http://graphics.stanford.edu/~seander/bithacks.html`

Memory efficiency woes

- 1. Minimize memory costs
  - Low program memory costs
  - Low data memory costs
- 2. Minimize memory transaction costs
  - Minimize on off chip swapping
  - Minimize data transfer between tasks
  - Minimize load and store
- It is usually hard to minimize both at the same time

Memory efficient

- Find algorithms use less on chip data memories. For example, some algorithms require fewer coefficients.
    - Trade computing complexity for memory efficiency
- Select algorithms with full memory access predictability if possible. Data can thus be stored in off-chip memory and pre-fetched efficiently when needed.

Discussion break: Memory space vs performance

- Assume you want to calculate x and y positions on the unit circle.
  - How would you do it if you need high performance?
  - How would you do it if you need low memory usage?

Example: Iterate over unit circle

```
for(i=0; i < 512; i++) {
    foo[i] *= cos((float)i*M_PI/256;
}
```

With lookup table for cos and sin

```
tmp = 0;
for(r = 0.0; r < M_PI*2; r+=M_PI/256){
    LUT[tmp++] = cos(r);
}

// ... at some other point in the program:
for(i=0; i < 512; i++){
    foo[i] *= LUT[i];
}
```

Vector rotation to calculate sequence of cos/sin values

```
// Calculate cosine function using vector rotation
A=0;
B=1;
for(i=0; i < 512; i++){
    foo[i] *= B;
    C = A*cos(M_PI/256)-B*sin(M_PI/256);
    B = A*sin(M_PI/256)+B*cos(M_PI/256);
    A=C;
}
```

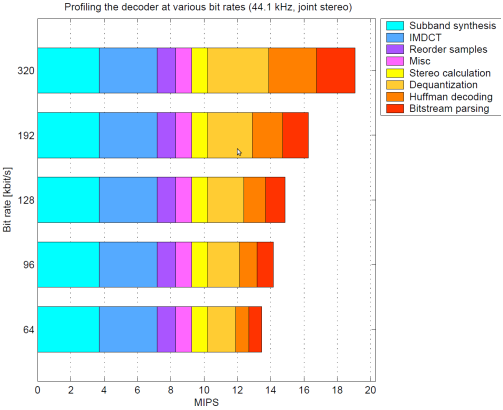
Vector rotation to calculate sequence of cos/sin values

- Not perfect
  - Precision is not that good
  - Calculation cost of vector rotation is higher than table lookup
    - Multiplication is fairly power hungry
    - (But we do not need power hungry memory accesss)
    - No need for fairly large lookup table
  - Only works for regular sin/cos function calls

Real time considerations

- In a real-time system it is important to know about worst case execution time (WCET)
- Different algorithms have different sensitivities to input data
- Program path analysis
  - Dynamic run time analysis
  - Static run time analysis

Profiling example of MP3 decoding



Coding quality checklist

- Try to use double precision instructions and keep computing inside the MAC
- Insert and optimize data measurement and scaling subroutines
- Use guard and shift together to avoid overflow
- Perform truncation and rounding at the right time

Important techniques for DSP firmware developer

- Using metrics like SNR and RMS error to determine the quality of the implementation
- Know how to calculate functions like `pow()`, `sin()`, `cos()`, `tan()`, etc efficiently in a given scenario
- Know how to minimize memory usage or trade off memory usage vs computing complexity
- Trade off latency vs throughput (c.f. lab 1)

HW verification

- Verification cost is huge (e.g., more than double the cost of RTL development)

HW verification

- Verification cost is huge (e.g., more than double the cost of RTL development)

Verification - Simple nonchecking testbench

```
initial begin
    rst = 1;
    #10; // Wait for 10 ns
    rst = 0;
    opa = 32;
    opb = 45;
    ctrl=1;
    #10;
    opa = 45;
    opb = 11;
    ctrl=2;
    #10;
    // And so on...
```

Verification - Simple nonchecking testbench

```
initial begin
    rst = 1;
    #10; // Wait for 10 ns
    rst = 0;
    opa = 32;
    opb = 45;
    ctrl=1;
    #10;
    opa = 45;
    opb = 11;
    ctrl=2;
    #10;
    // And so on...
```

Not a good idea!

Verification using files

```
initial begin
  fd = $fopen("indata","r");
  while(!finished) begin
    @(posedge clk);
    $fgets(buf,fd);
    numdata= $sscanf(line,"%08x␣%08x␣%08x",x,y,z);
    if(numdata == 3) begin
      opa <= x;
      opb <= y;
      if(outdata !== z) begin
        $display("Output␣data␣incorrect!");
        $stop;
      end
```

Verification using files

- Essentially what we do in the labs
- Very nice for processor development
    - You need a simulator anyway for processor development
- Can be cumbersome for many other systems where it is not natural to write a simulator

(System)Verilog and VHDL are programming languages!

- Write testbenches in a structured manner
  - Divide and conquer!
  - Use tasks/procedures to make the testbenches easier to understand
  - You can, in many cases, make the testbench selfchecking without any need for external files
  - Model the system using behavioral code

Example: Verifying a divider

```
task test_divs; // (Would be a procedures in VHDL)
  input [BITS1:0] dividend;
  input [BITS1:0] divisor;
  begin
    @(posedge clk);
    divop <= `SIGNED;
    divopa <= dividend;
    divopb <= divisor;
    ctrlstartdiv <= 1;
    @(posedge clk);
    while(div_flag_busy) begin
      @(posedge clk);
    end
```

Example: Verifying a divider

```
task simpletest;
begin
  $display("Testing␣simple␣values");
  test_divu(1,1);
  test_divu(2,1);
  test_divu(2,2);
  $display("Testing␣corner␣values␣(large)");
  test_divu(32'h80000000,1);
  test_divu(32'h80000000,32'h80000000);
  test_divu(0,32'h80000000);
  test_divu(1,32'h80000000);
  test_divu(2,32'h80000000);
  // And so on
```

Example: Verifying a divider

```
initial begin
  startclock();
  releasereset();
  simpletest();
  simpletest_signed();
  test_small_values();
  test_corners();
  test_random_values();
  stopclock();
  $display("All␣tests␣finished!");
end
```

Design for verification

- Remove unneeded complexity $\Rightarrow$ Reduced verification cost

Other things to look into

- SystemVerilog has a lot of nice features for testbenches
  - Fifos, classes, interfaces, assertions and many others
  - Look for Verification Methodology Manual for inspiration

Oscar Gustafsson

www.liu.se

LINKÖPING UNIVERSITY