04 - DSP Architecture and Microarchitecture

Andreas Ehliar

September 11, 2014
Conclusions - Instruction set design

- An assembly language instruction set must be more efficient than Junior
- Accelerations shall be implemented at arithmetic and algorithmic levels.
- Addressing and data accesses can be executed in parallel with arithmetic computing.
- Program flow control, loop or conditional execution, can also be accelerated
Conclusions - Instruction set design

- A DSP processor will seldomly have a pure RISC-like instruction set
- To accelerate important DSP kernels, CISC-like extensions are acceptable (especially if they don’t add any real hardware cost)
  - (Also, note that both RISC and CISC are losers in the processor wars today, real processors are typically hybrids)
What if you can’t create an ASIP?

- Trade program memory for performance
  - To avoid control complexity (loop unrolling)
  - To avoid addressing complexity

- Other clever programming tricks
  - Conditional execution
  - (Self modifying code)
  - Rewrite algorithm
  - etc. . .
History of DSP architectures

- Von Neumann architecture vs Harvard architecture

[Liu2008, Figure 3.3]
History of DSP architectures

[Liu2008, Figure 3.4]

- a) Normal Harvard architecture
- b) Words from PM can be sent to the datapath
- c) Use a dual port data memory
History of DSP architectures

Efficient FIR filter with only two memories (PM and DM)

Alt 1: Carries coefficient as immediate

Alt 2: Override instruction fetch, fetch data from PM

```
mac A, DM[AR0++%], -1
mac A, DM[AR0++%], -743
mac A, DM[AR0++%], 0
mac A, DM[AR0++%], 8977
mac A, DM[AR0++%], 16297
mac A, DM[AR0++%], 8977
mac A, DM[AR0++%], 0
mac A, DM[AR0++%], -743
mac A, DM[AR0++%], -1
rnd A
```

```
conv 8,DM[AR0++%]
.data -1
.data -743
.data 0
.data 8977
.data 16297
.data 8977
.data 0
.data -743
.data -1
rnd A
```

More orthogonal

No need for wide PM
History of DSP architectures

▶ d) Use a small (loop?) cache to allow for one memory to be shared between PM and DM
▶ e) Typical three memory configuration.

[Liu2008, Figure 3.5]
DSP Processor vs DSP Core

[Liu2008, Figure 3.2]
Architecture selection

- Selecting a suitable ASIP architecture for the desired application domain
- The decision includes how many function modules are required, how to interconnect these modules (relations between modules), and how to connect the ASIP to the embedded system
- Closely related to instruction set selection if an efficient implementation is desired
DSP processor developers have an advantage over general purpose CPU developers (e.g. Intel, AMD, ARM):

- Known applications
- Known scheduling requirements
- Vector based algorithms and processing
Challenges of DSP parallelization

- Hard real time and high performance
- Low memory and low power costs
- Data and control dependencies

Remember Amdahl’s law: Your speedup is ultimately limited by the amount of sequential parts you have in your application.
Ways to speed up a processor - Discussion break

- Programmer visible:
  - VLIW
  - Multiple memories
  - Accelerators
  - SIMD

- Programmer invisible:
  - Cache
  - Pipelining
  - Superscalar (in- or out-of-order)
  - Dataforwarding
  - Voltage scaling
  - Branch prediction
  - Multiple clocks
  - Low-level circuit optimizations
Advanced architectures: Dual MAC

[Liu2008, Figure 3.22]

Liu2008
Advanced architectures: Dual MAC

- Allows you to speed up operations such as FIR filters.
- Can allow you to calculate \( y[n] = \sum_{k=0}^{N-1} h[k]x[n - k] \) and \( y[n + 1] = \sum_{k=0}^{N-1} h[k]x[n + 1 - k] \) at the same time for example.
  - Note: Will roughly halve the number of memory accesses (More on this in a later lecture.)
Advanced architectures: SIMD

Program memory carries only one instruction

I-decoding

Address
Execution unit

Address
Execution unit

Address
Execution unit

Address
Execution unit

[Liu2008, Figure 3.24 (modified)]

- Advantage: Low power and area
- Disadvantage: Difficult to use efficiently, very difficult target for a compiler.
Advanced architectures: VLIW

- **Why:** DSP tasks are relatively predictable
  - A parallel datapath gives higher performance
- **How:** Very Large Instruction Word
  - Multiple instruction issues per-cycle
  - Compiler manages data dependency
- **Challenges**
  - Memory issue and on chip connections
  - Register (fan-out ports) costs
  - Hard compiler target
Advanced architectures: Superscalar

- Analyze instruction flow
- Run several instruction in parallel
  - (And possibly out of order)
VLIW vs Superscalar

▶ VLIW:
  ▶ Relatively easy to design and verify the hardware
  ▶ Not code efficient due to instruction size and NOP instructions
  ▶ Hard to keep binary compatibility
  ▶ Hard to create an efficient compiler

▶ Superscalar
  ▶ Hard to design and verify the hardware
  ▶ Good code efficiency, relatively small instructions, No NOPs needed
  ▶ Easier to manage compatibility between processor versions
Multicore architectures

- Heterogenous or homogenous
  - Well known heterogenous architecture: Cell
  - Well known homogenous architecture: Modern X86
- Usually harder to program than single threaded arch.
- Heterogenous architectures are well suited for ASIPs
  - Standard MCU for main part of application
  - Specialized DSP for performance critical parts
Summary: Advanced Architectures

- Dual MAC: Easy, not a huge improvement
- SIMD DSP: Very good for regular tasks
- VLIW: Good parallelism but hard for compiler
- Superscalar: Relatively easy for a compiler, but highest silicon cost and verification cost
- Multicore: Whenever a single core is not powerful enough
Summary: Advanced Architectures

[Liu2008, Figure 4.5 (modified)]

Liu2008
ASIP Design flow

- Understand target application
- Design architecture and assembly instruction set
- Create microarchitecture specification
- Write RTL code
- Synthesize code
- Backend design
- Tape out
- Celebrate!
ASIP Design flow

Application / function coverage analysis

Instruction set proposal and 90% 10% locality

Instruction set simulator and assembler

Benchmarking: speed up and coverage

No

satisfied

yes

Release the instruction set architecture

yes

Micro architecture design, RTL, and VLSI

Source code profiling

Instruction set design

Processor implementation
Understanding applications

- Read standards
- Read and profile reference code
- Read research papers about target application
  - IEEE Xplore, Scopus, Google Scholar, etc
- Interview application expert
90-10 code locality rule

- About 10% of the instructions are used about 90% of the time
  - (Not really a rule, more of an observation)
  - Holds fairly well for DSP-like applications
- We should create an instruction set so that those 10% can be executed efficiently
Profiling

- Pen and paper method
  - Look at reference code (Matlab, C, pseudo code etc)
  - Manually figure out required number of
  - Arithmetic operations
  - Control flow instructions
  - Memory accesses
  - Address calculations
- Works for small applications/subroutines
  - Such as the problems in the exam
Profiling tools

- Tell compiler that you want to use profiling
  - For gcc: -pg
- Signal (timer interrupt) is generated regularly
- Every 10 ms when using gcc on Linux
  - Current PC is stored
- Functions are instrumented
  - A counter is incremented for each function call
## Profiling example: mplayer using gprof in Linux

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self</th>
<th>self calls</th>
<th>ms/call</th>
<th>ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12.40</td>
<td>2.27</td>
<td>2.27</td>
<td>73363735</td>
<td>0.00</td>
<td>decode_residual</td>
</tr>
<tr>
<td>11.86</td>
<td>4.44</td>
<td>2.17</td>
<td>6180354</td>
<td>0.00</td>
<td>put_h264_qpel8o</td>
</tr>
<tr>
<td>8.14</td>
<td>5.93</td>
<td>1.49</td>
<td>21857226</td>
<td>0.00</td>
<td>h264_h_loop_fil</td>
</tr>
<tr>
<td>7.43</td>
<td>7.29</td>
<td>1.36</td>
<td>9922560</td>
<td>0.00</td>
<td>ff_h264_decode_</td>
</tr>
<tr>
<td>5.52</td>
<td>8.30</td>
<td>1.01</td>
<td>18181724</td>
<td>0.00</td>
<td>put_h264_chroma</td>
</tr>
<tr>
<td>4.70</td>
<td>9.16</td>
<td>0.86</td>
<td>82688</td>
<td>0.01</td>
<td>loop_filter</td>
</tr>
<tr>
<td>4.67</td>
<td>10.02</td>
<td>0.86</td>
<td>9922560</td>
<td>0.00</td>
<td>ff_h264_filter_</td>
</tr>
<tr>
<td>4.43</td>
<td>10.83</td>
<td>0.81</td>
<td>23096042</td>
<td>0.00</td>
<td>h264_h_loop_fil</td>
</tr>
</tbody>
</table>
Profiling flow for ASIP designers

- Find reference application code
- Compile it for your desktop computer
  - With profiling/debugging information
  - Run it with typical input data
  - Look at profiling output to quickly determine parts that are likely to be critical
Profiling pitfalls

- Is your reference code well optimized?
- Is your reference code using hardware features in such a way that your profiling output is misleading?
  - Hand optimized assembler code
  - Memory subsystem, cache size
  - Specialized instructions
  - Vector instructions
Profiling pitfalls

- While care must be taken in interpreting the results, profiling is still a very good friend of an ASIP designer!
- Never optimize your code unless you have profiled it and seen that it makes sense to optimize it!
  - "The First Rule of Program Optimization: Don’t do it. The Second Rule of Program Optimization (for experts only!): Don’t do it yet.” — Michael A. Jackson
Advanced profiling

- Profiling of other kinds of events:
  - Cache hits and cache misses
  - Floating point operations
  - Taken/not taken branches
  - Predicted/mispredicted branches
  - TLB misses
  - Instruction decoder stalls
  - Etc...

- Interested? Look at oprofile or VTune
Once you have an assembler and instruction set simulator for your ASIP you can benchmark your own ASIP

Count the number of times each instruction is used in a number of applications

Useful for fine tuning of the DSP architecture but too time consuming to start with

Without a compiler it is very hard to profile complete applications

See lab 4!
Static vs Dynamic Profiling

- Dynamic profiling: Running an application with typical input data
- Static profiling: Analyzing the control flow graph of an application and determining worst case execution time (WCET)
- Critical for hard real time applications!
  - Some hardware features complicates this:
    - Interrupts, caches, superscalar, OoO, branch prediction, DMA, etc...
Evaluating instruction sets

- Evaluation of an instruction set
  - Cycle cost and memory usage
  - Suitability for specific applications
- How to evaluate a processor
  - Good assembly instruction set
  - Good (open and scalable) architecture
  - (Max clock frequency, low power, less area)
- Use benchmarking techniques!
General benchmarks

- Algorithm benchmarks/kernel benchmarks
- Normal precision and native word length
- What to check:
  - Cycle costs of kernels, prologs, and epilogs
  - Program/data memory costs
- Algorithms including
  - FIR, IIR, LMS, FFT, DCT, FSM
Third Party Benchmarks

- BDTI: Berkeley Design Tech Incorporation
  - Professional hand written assembly
  - http://www.bdti.com

- EEMBC (the EDN Embedded Microprocessor Benchmark Consortium), fall into five classes:
  - automotive/industrial, consumer, networking, office automation, and telecommunication
  - http://www.eembc.org
Ideal benchmark

- The application you are interested in!
- Preferably optimized for your DSP architecture
  - Difficult in practice
Step 1: Partition each assembly instruction into microoperations, allocate each microoperation into corresponding hardware modules.

Step 2: Collect all microoperations allocated in a module and specify **hardware multiplexing** for RTL coding of the module.

Step 3: Fine-tune intermodule specifications of the ASIP architecture and finalize the top-level connections and pipeline.
Hardware Multiplexing

- Reusing one hardware module for several different operations
- Example: Signed and unsigned 16-bit multiplication
Possible functions
1. A + C
2. A + D
3. B + C
4. B + D
5. A * C
6. A * D
7. B * C
8. B * D
9. SAT(A + C)
10. SAT(A + D)
11. SAT(B + C)
12. SAT(B + D)
13. SAT(A * C)
14. SAT(A * D)
15. SAT(B * C)
16. SAT(B * D)

[Liu2008]
Hardware multiplexing can be implemented either by SW or by configuring the HW.

A processor is basically a very neat design pattern for multiplexing different HW units.

Perhaps the most important skill of a good VLSI designer.
Typical design pattern for datapath modules

[Liu2008]
Discussion break:

- What is most area expensive of these units?
  - 17 x 17 bit multiplier
  - 32-bit Adder/subtracter
  - 32-bit 16 to 1 mux
  - 32-bit Adder
  - 8 KiB memory
Relative areas of a few different components

- 32-bit Adder: 0.2 to 1 area units
- 32-bit Adder/subtractor: 0.3 to 2 area units
- 32-bit 16 to 1 mux: 0.5 – 0.6 area units
- 17 x 17 bit multiplier: 1.3-3.7 area units
- 8 KiB memory (32 bit wide): 33 area units
Performance properties

- Relative maximum frequencies
  - 32-bit adder: 0.1 to 1
  - 32-bit adder/subtractor: 0.1 to 0.9
  - 32-bit 16 to 1 mux: 0.31 to 0.9
  - 17 x 17 bit multiplier: 0.11 – 0.44
  - 8 KiB memory (32 bit wide): 0.53
Optimizing memory size is often the most important task

- MP3 decoder example
- All memories in the chip are 3 time the size of the DSP core itself
- (I/O pads are also larger than the DSP core itself)
Microarchitecture design of an instruction

- Required microoperations for a typical convolution instruction:
  - conv ACRx, DM0[ARy%++] , DM1(ARz++)

- Required microoperations:
  - Instruction decoding
  - Perform addressing calculation
  - Read memories
  - Perform signed multiplication
  - Add guard bits to the result of the multiplication
  - Accumulate the result
  - Set flags
  - For a combined repeat/conv instruction:
    - PC <= PC while in the loop
    - PC <= PC + 1 as the last step in the loop
    - No saturation/rounding during the iteration
    - Saturate/round after final loop iteration
The register file (RF)

- The RF gets data from data memories by running load instructions while preparing for an execution of a subroutine.
- While running a subroutine, the register file is used as computing buffers.
- After running the subroutine, results in the RF will be stored into data memories by running store instructions.
Connected to almost all parts of the core
Register file schematic

Write circuit

from register file
from memory 1
from memory 2
from ALU
from ports
.....
from MAC

ctrl_reg_in

Store circuit

register 1
register 2
register 3
register n

Read circuit

register 1
register 2
register 3
register n

......

OP A
OPB
ctrl_o_a
ctrl_o_b
OPA
OPB

[Liu2008]
Register file area comparisons

- Adder (for comparison): 0.1 - 1
- One port, both read/write: (uncommon)
  - Relative area: 2.5-2.7
- One read port, one write port: Accumulator registers
  - Relative area: 2.5-2.6
- Two read ports, one write: Basic RISC RF
  - Relative area: 3-3.2
- Four read ports, two write: Simple VLIW, Simple superscalar
  - Relative area: 4.6-5.8
Register file speed

- Almost (but not quite) the same speed as a very fast 32-bit adder (in this particular technology)
- Also note that it is possible to use special register file memories (but at an increased verification cost)
A processor architect has to decide how the register file should work when reading and writing the same register.

- **Read before write**
  - The old value is read.

- **Write before read**
  - The new value is read (more costly in terms of the timing budget).
Physical design: fan-out problem

Fan-out of the control signal
For the first stage: $16 \times 16 \times 2 = 512$

From 32 registers in a register file

Fan-out of the control signal
For the second stage: $16 \times 8 \times 2 = 256$

Fan-out of the control signal
For the third stage: $16 \times 4 \times 2 = 128$

Fan-out of the control signal
For the fourth stage: $16 \times 2 \times 2 = 64$

Fan-out of the control signal
For the fifth stage: $16 \times 1 \times 2 = 32$

Selected operand

[Liu2008]
Register File in Verilog

```verilog
reg [15:0] rf[31:0]; // 16 bit wide RF with 32 entries
always @(posedge clk) begin
    if(we) rf[waddr] <= wdata;
end

always @* begin
    op_a = rf[opaddr_a];
    op_b = rf[opaddr_b];
end
```