04 – DSP Architecture and Microarchitecture

Oscar Gustafsson

Conclusions - Instruction set design

- C hides memory addressing costs and loop costs
- At assembly language level, memory addressing must be explicitly executed.
- We can conclude that most memory access and addressing can be pipelined and executed in parallel behind running the arithmetic operations.

Conclusions - Instruction set design

- One essential ASIP design technique will be grouping the arithmetic and memory operations into one specific instruction if they are used together all the time
- Remember this during lab 4!

Conclusions - Instruction set design

- To hide the cost of memory addressing and data access is to design smart addressing models by finding and using regularities of addressing and memory access.
- Addressing regularities:
  - postincremental addressing
  - modulo addressing
  - postincremental with variable step size
  - and bit-reversed addressing.

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

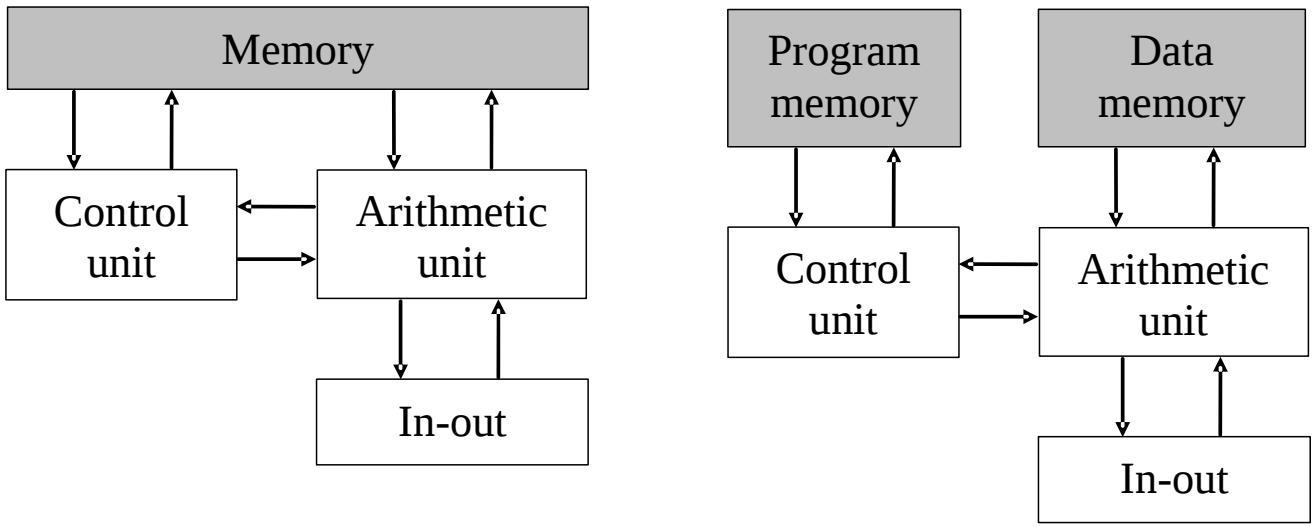
- A DSP processor will seldomly have a pure RISC-like instruction set
- To accelerate important DSP kernels, CISC-like extensions are acceptable (especially if they don't add any real hardware cost)
  - (Also, note that both RISC and CISC are losers in the processor wars today, real processors are typically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

What if you can't create an ASIP?

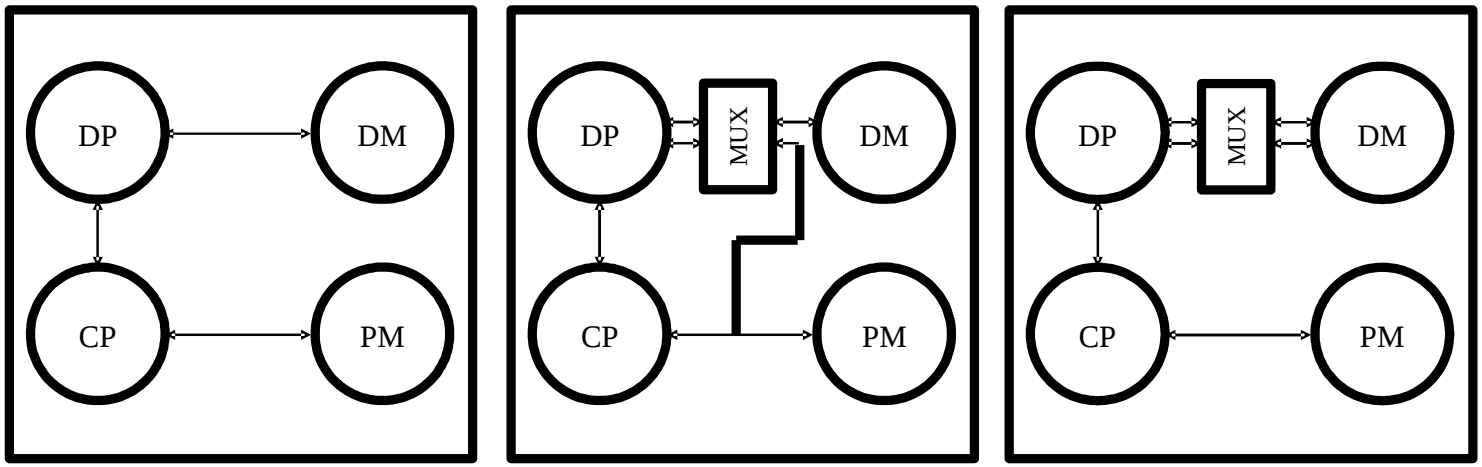
- Trade program memory for performance
  - To avoid control complexity (loop unrolling)
  - To avoid addressing complexity
- Other clever programming tricks
  - Conditional execution
  - (Self modifying code)
  - Rewrite algorithm
  - etc…

History of DSP architectures

- Von Neumann architecture vs Harvard architecture



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

```
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
```

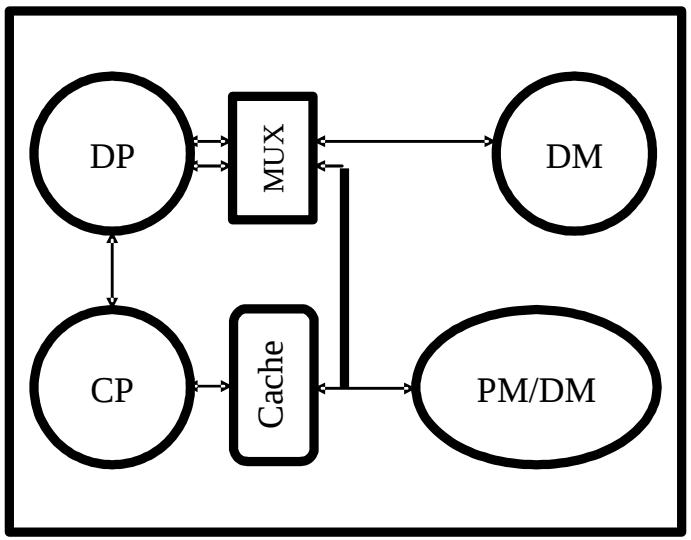
More orthogonal

Alt 2: Override instruction fetch, fetch data from PM

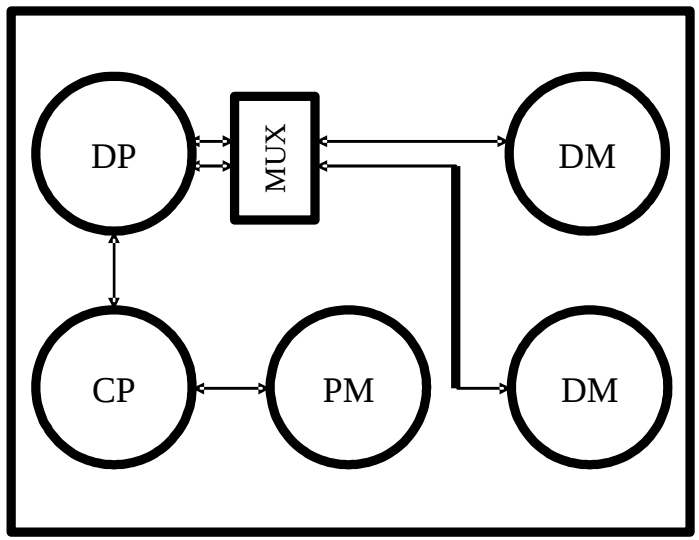
```
conv  8,DM[AR0++%]
.data -1
.data -743
.data 0
.data 8977
.data 16297
.data 8977
.data 0
.data -743
.data -1
rnd   A
```

No need for wide PM

History of DSP architectures



(d)

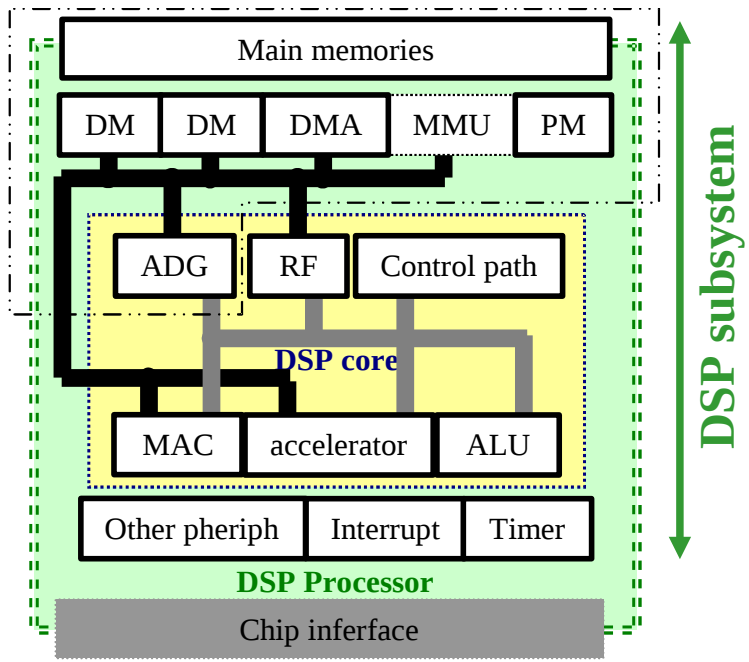


(e)

[Liu2008, Figure 3.5]

- d) Use a small cache to allow for one memory to be shared between PM and DM
- e) Typical three memory configuration

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

Architecture selection

- 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

Architecture selection

- 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 parts that cannot be sped up.

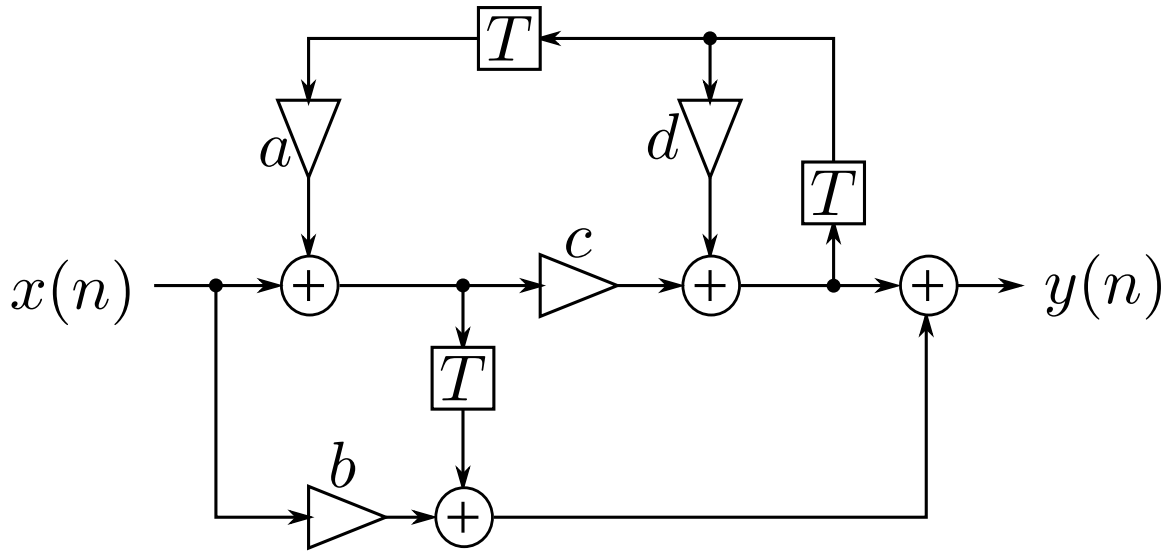
Ways to speed up a processor - Discussion break

- Programmer visible:
  - VLIW
  - Multiple memories
  - Accelerators
  - SIMD
  - Multicore
- Programmer invisible
  - Cache
  - Pipelining
  - Superscalar (in- or out-of-order)
  - Dataforwarding
  - Branch prediction

Parallelism in DSP Algorithms

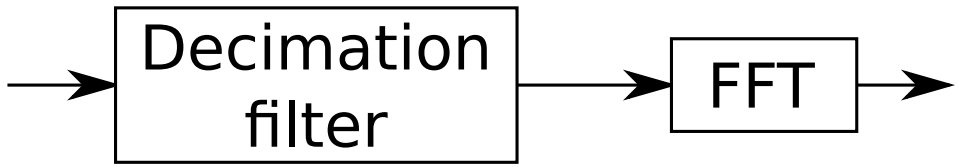
- Parallelism can be seen on several different abstractions levels
- Different types of parallelism can be used differently

Parallelism in DSP Algorithms – Operation level



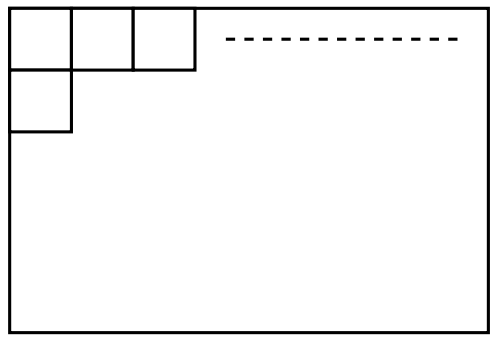
- Some operations can be performed in parallel

Parallelism in DSP Algorithms – Algorithm/block level



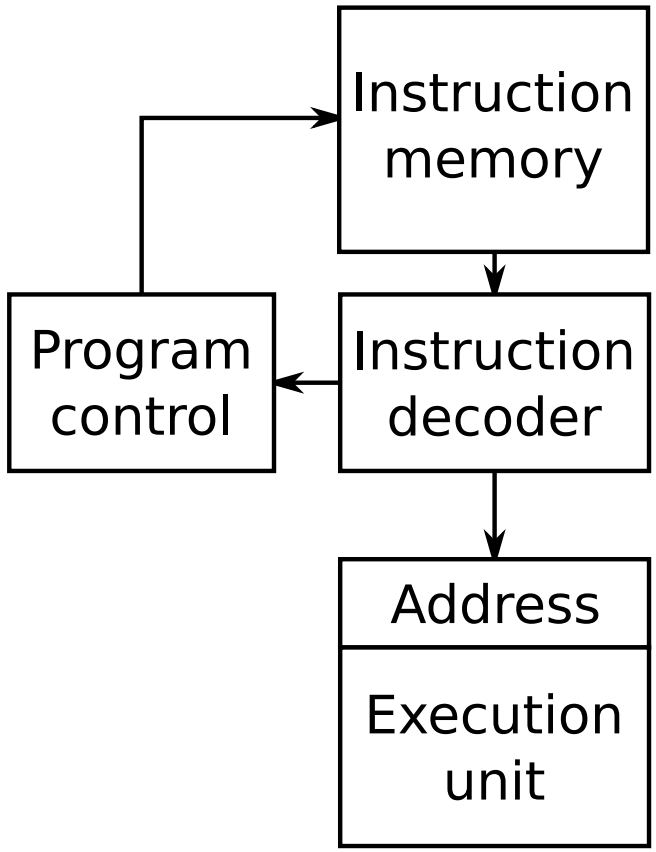
- Algorithms/blocks can be computed in parallel
- Not fundamentally different compared to operation level

Parallelism in DSP Algorithms – Data level

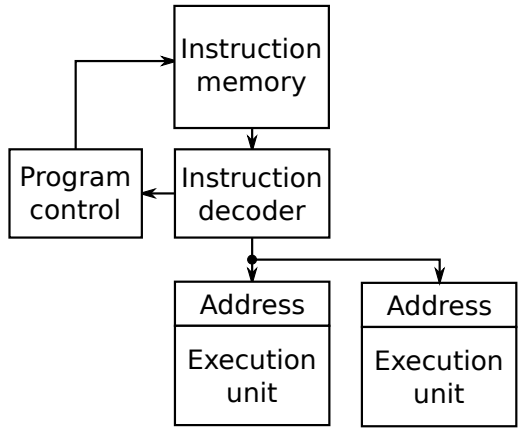


- Different data (parts of image in this case) can be computed in parallel

Standard DSP architecture

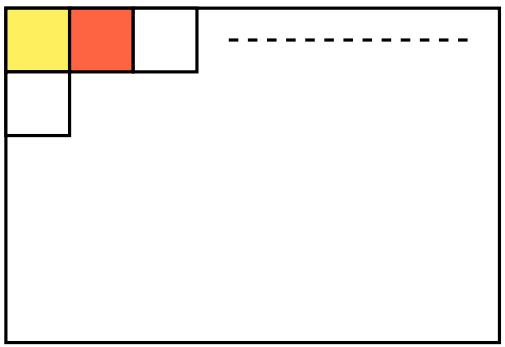


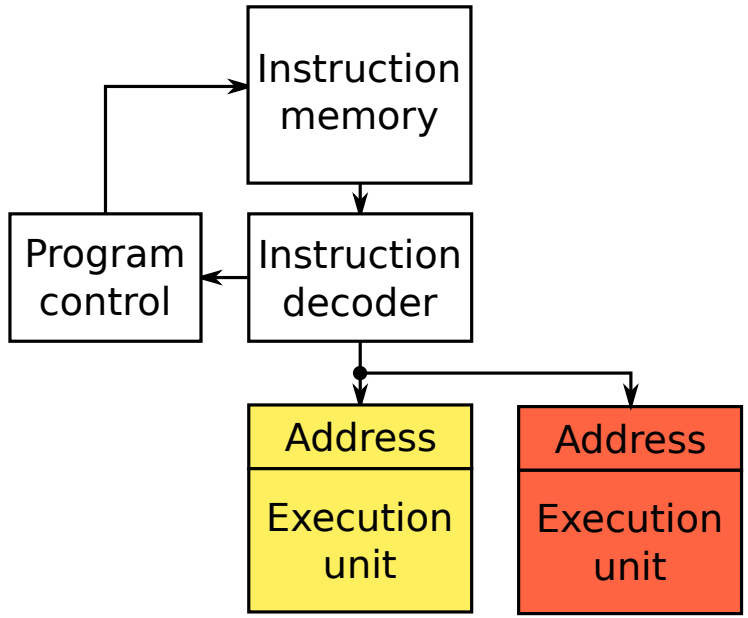
Advanced architectures: SIMD



- Single instruction, multiple data
- Advantage: low power and area
- Disadvantage: difficult to use efficiently, very difficult target for a compiler.

Advanced architectures: SIMD

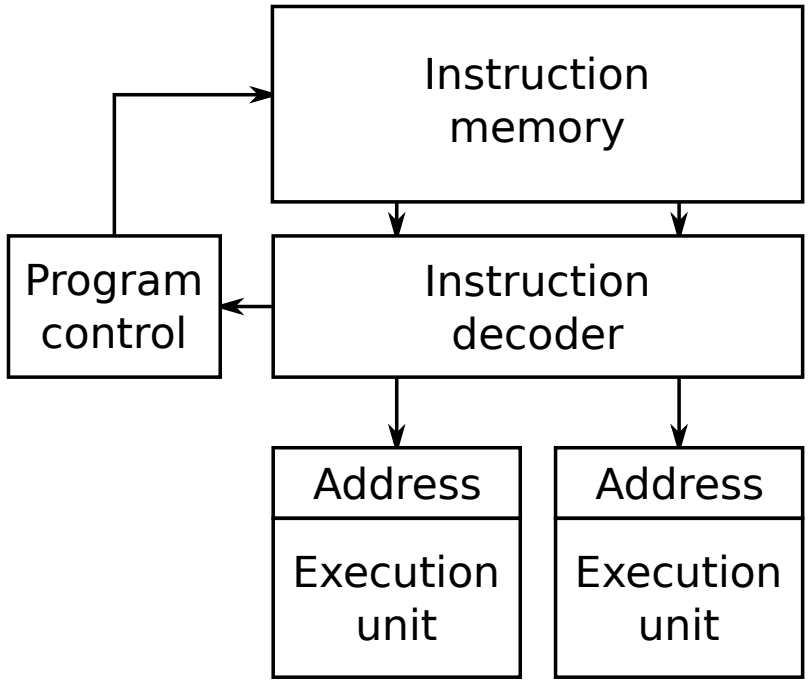




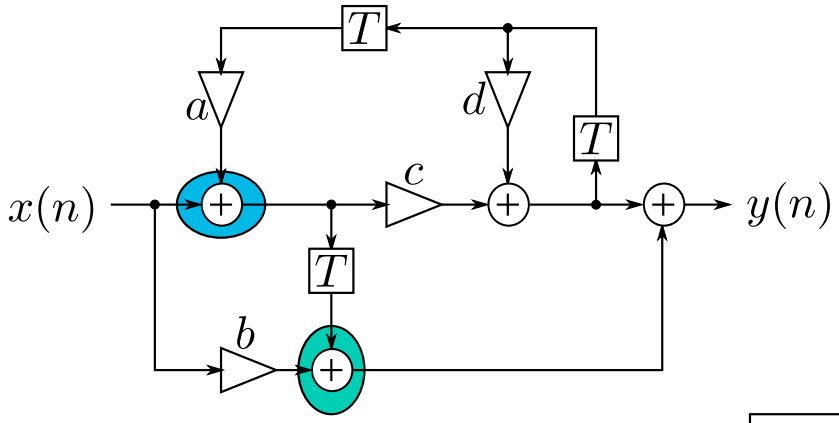
Advanced architectures: VLIW

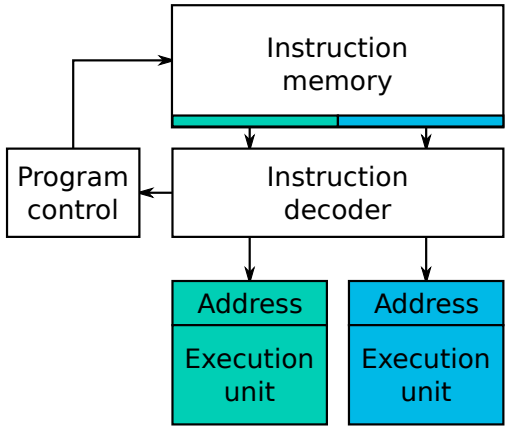
- Why: DSP tasks are relatively predictable
  - A parallel datapath gives higher performance
- How: Very Large Instruction Word
  - Multiple instruction issues per-cycle
  - Compiler manages data dependency
- Challenges
  - Memory issue and on chip connections
  - Register (fan-out ports) costs
  - Hard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

Advanced architectures: VLIW

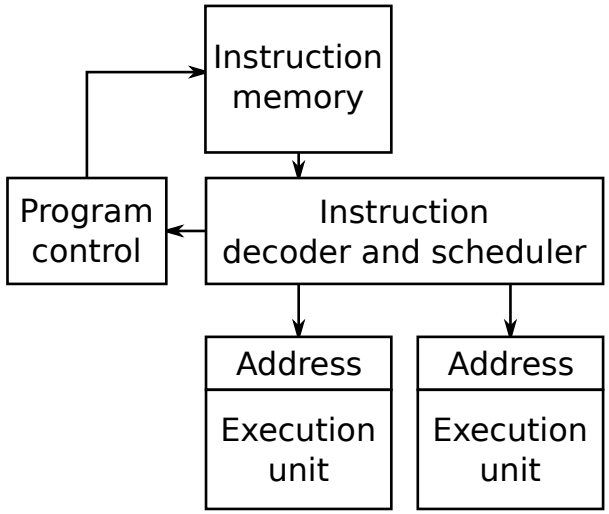


Advanced architectures: VLIW



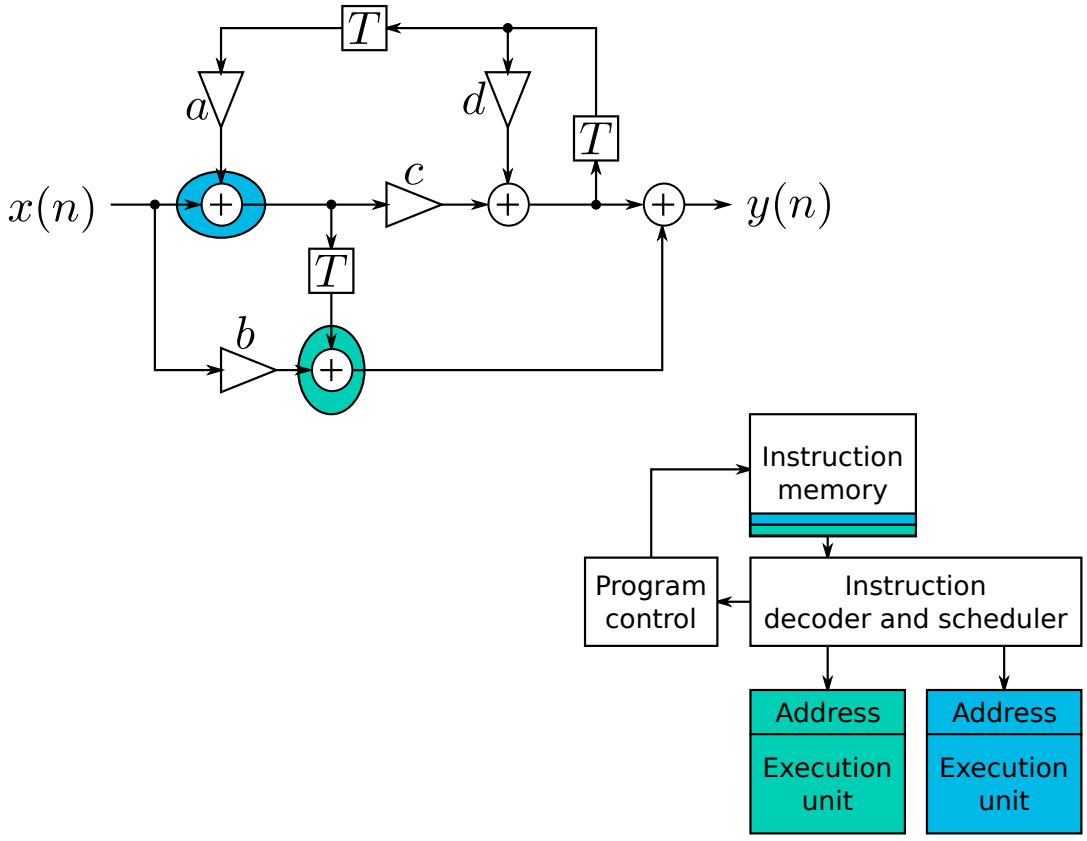


Advanced architectures: Superscalar



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

Advanced architectures: Superscalar



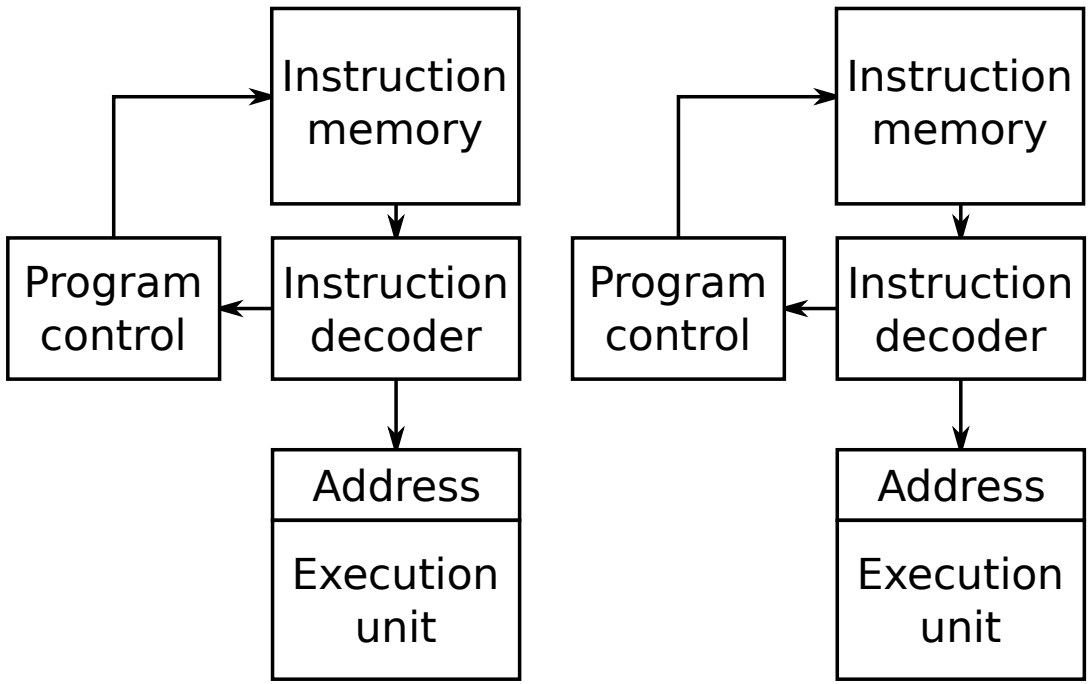
VLIW vs Superscalar

- 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

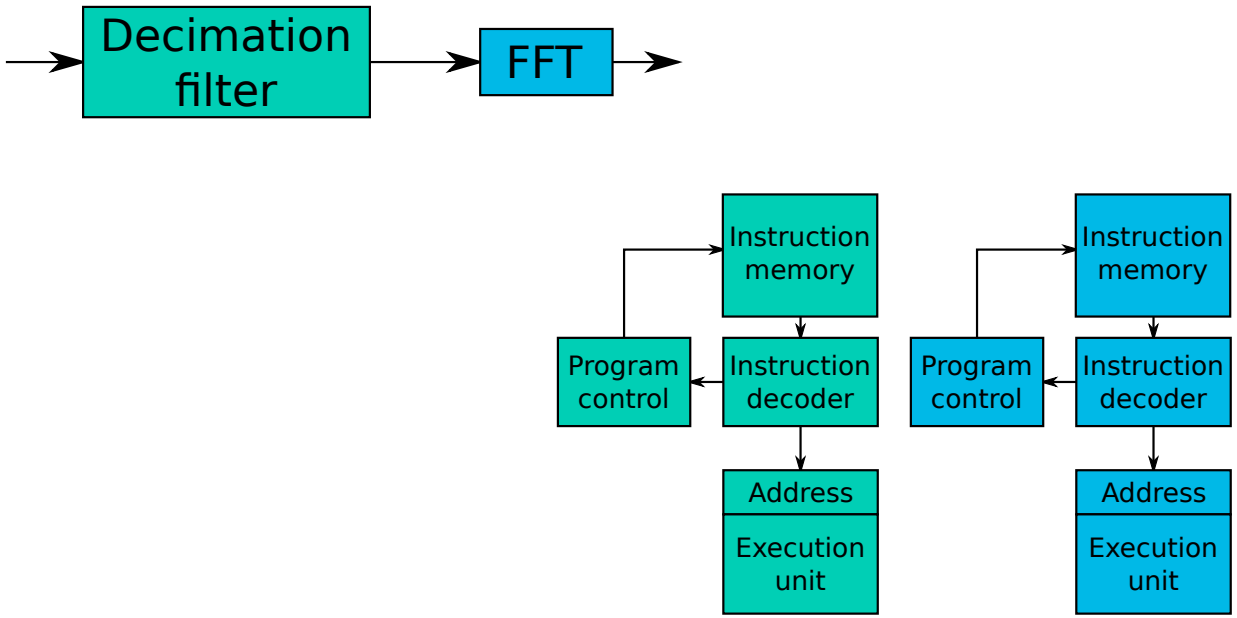
Multicore architectures

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

Advanced architectures: Multicore



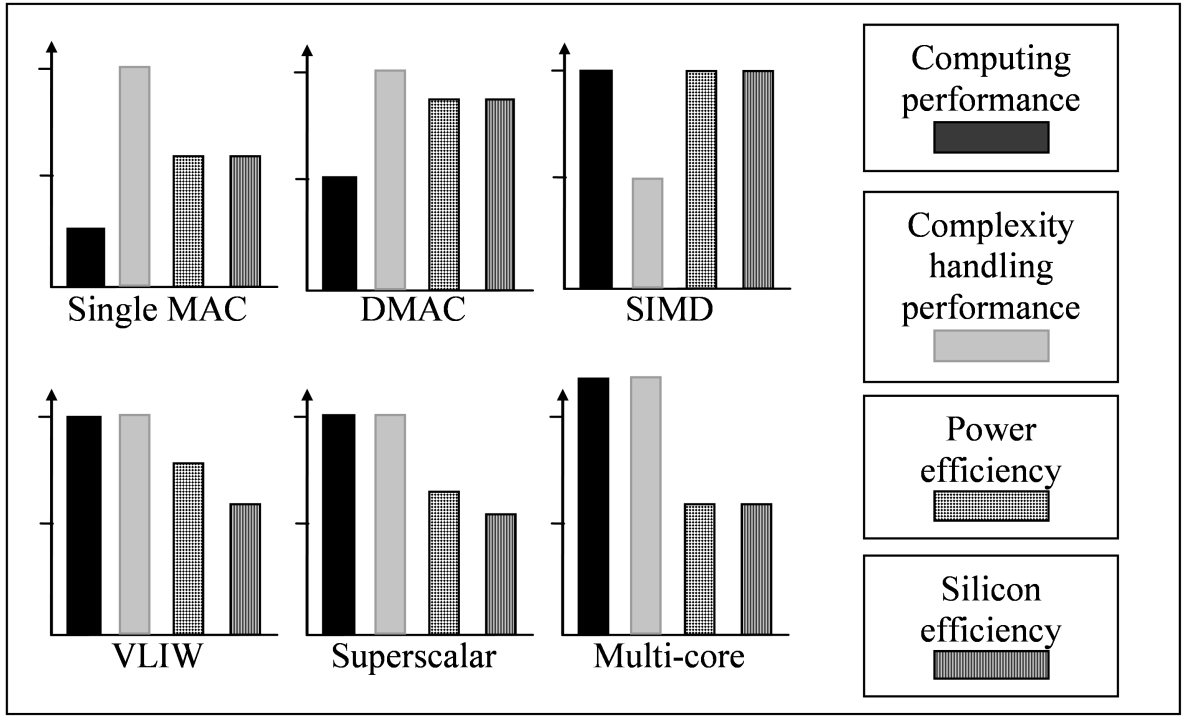
Advanced architectures: Multicore



Summary: Advanced Architectures

- 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
- You can choose any (combination of) architecture(s) which makes sense

Summary: Advanced Architectures

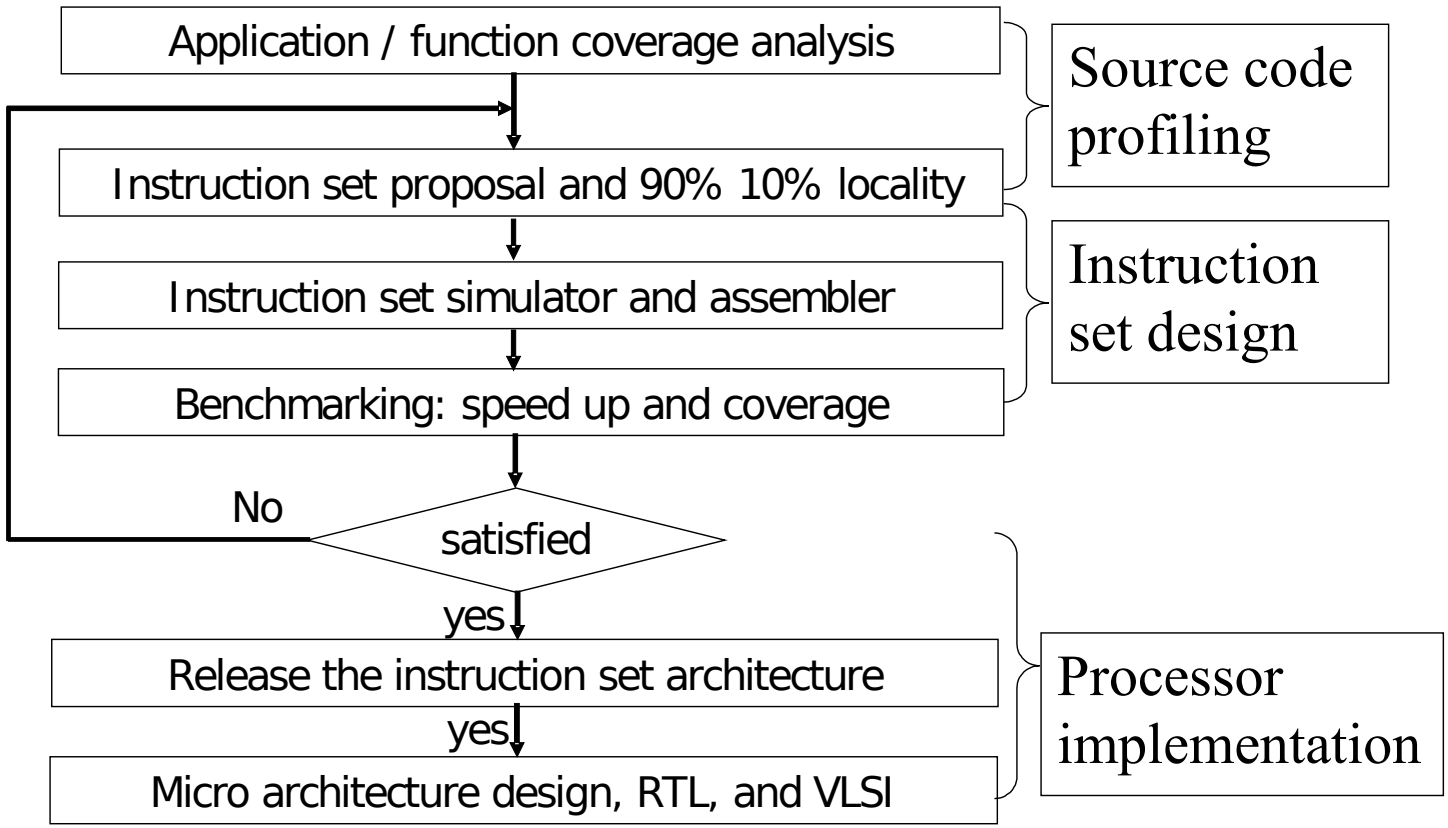


[Liu2008, Figure 4.5 (modified)]

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



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.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 12.40      2.27     2.27 73363735     0.00     0.00  decode_residual
 11.86      4.44     2.17  6180354     0.00     0.00  put_h264_qpel8o
  8.14      5.93     1.49 21857226     0.00     0.00  h264_h_loop_fil
  7.43      7.29     1.36  9922560     0.00     0.00  ff_h264_decode_
  5.52      8.30     1.01 18181724     0.00     0.00  put_h264_chroma
  4.70      9.16     0.86    82688     0.01     0.07  loop_filter
  4.67     10.02     0.86  9922560     0.00     0.00  ff_h264_filter_
  4.43     10.83     0.81 23096042     0.00     0.00  h264_h_loop_fil
```

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

ASIP profiling

- 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

Oscar Gustafsson

www.liu.se

LINKÖPING UNIVERSITY