; Reality check: Data hazards!
; Assembler code v3:
  repeat 256,endloop
  load r0,DM1[DM0[ptr0++]]
  store DM0[ptr1++],r0
endloop:

// 512 clock cycles

➤ Short discussion break: How to rewrite the code above to avoid the data hazard?
Memory indirect addressing

; Assembler code v4:
    repeat 64,endloop
    load r0,DM1[DM0[ptr0++]]; Unrolls the loop
    load r1,DM1[DM0[ptr0++]]; to avoid data
    load r2,DM1[DM0[ptr0++]]; hazards
    load r3,DM1[DM0[ptr0++]]
    store DM0[ptr1++],r0
    store DM0[ptr1++],r1
    store DM0[ptr1++],r2
    store DM0[ptr1++],r3
endloop:
Alternative 2: Memory indirect addressing during store

; Assembler code v5:
  repeat 256,endloop
  load r0,DM0[ptr0++]
  store DM0[ptr1++],DM1[r0] // DM0 = DM1[r0]
endloop:

// 512 clock cycles

- Justification: It is better if the pipeline is created in such a way that a store takes a long time to complete than a load.
  - (A store will seldom generate data dependencies whereas a load to a register will easily generate data dependencies as seen in the first alternative.)
Alternative 2: Memory indirect addressing during store (unrolled)

; Assembler code v6:
repeat 64,endloop
load r0,DM0[ptr0++]
load r1,DM0[ptr0++]
load r2,DM0[ptr0++]
load r3,DM0[ptr0++]
// A store buffer can simplify some
// of the data hazards here.
// (might still need some unrolling)
store DM0[ptr1++],DM1[r0]
store DM0[ptr1++],DM1[r1]
store DM0[ptr1++],DM1[r2]
store DM0[ptr1++],DM1[r3]
endloop:
Alternative 3: Rewrite loop as follows

- Output stored in DM1 this time around, remaining data in DM0

; Assembler code v7:
  load r0,DM0[ptr0++]

repeat 255,endloop
load r0,DM0[r0]
store DM1[ptr1++], r0
load r0,DM0[ptr0++]
endloop
load r0,DM0[r0]
store DM1[ptr1++], r0

// 768 clock cycles for loop, no improvement (yet)
Alternative 3: Merge instructions

- No real data dependency between the marked instructions, merge these into one!

; Assembler code v7:

    load  r0,DM0[ptr0++]

repeat 255,endloop

    load  r0,DM0[r0]

    store DM1[ptr1++],r0 // These two instructions

    load  r0,DM0[ptr0++] // can be merged without

    store DM1[ptr1++], r0  // additional HW cost!

endloop

    load r0,DM0[r0]

    store DM1[ptr1++], r0

// 768 clock cycles for loop, no improvement (yet)
Alternative 3: Merge instructions

- A form of software pipelining has been used here
  - (The inner loop operates partly on iteration i, and partly on iteration i+1)

; Assembler code v8:
load r0,DM0[ptr0++]  // Prologue

repeat 255,endloop
load r0,DM0[r0]
loadstore r0,DM0[ptr0++], DM1[ptr1++],r0  // These two instructions

// loadstore does the following:
// DM1[ptr1++] = r0; r0 = DM0[ptr0++]

endloop
load r0,DM0[r0]  // Epilogue
store DM1[ptr1++], r0

768 clock cycles for loop, no improvement (yet)
Alternative 3: Rewrite loop as follows

- Advantage of alternative 3:
  - The pipeline depth of loadstore is the same as the pipeline depth of load and store
  - The instruction may also be useful in other situations such as when copying values from one memory to another
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!
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

[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
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

More orthogonal

mac A, DM[AR0++%], -1  
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

[Figure 3.5]

- d) Use a small (loop?) 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
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 amount of sequential parts you have in your application.
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
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

[Li2008, 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