LECTURE #3
Micro-Processor Core

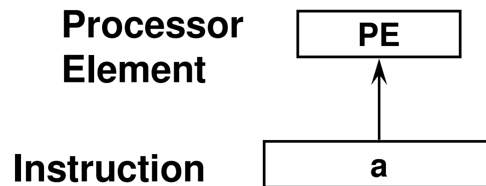
- Micro-Processor Core
  - Overview
  - Scheduling
- VLIW / EPIC
- Out-Of-Order

References of the day

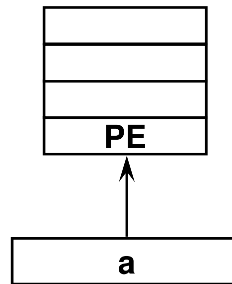
- “Computer Architecture - A Quantitative Approach” (The second edition), John L. Hennessy, David A. Patterson, Chapter 3-4 (p. 125-370)
- “Computer Organization and Design”, John L. Hennessy, David A. Patterson, Chapter 5-6, 9 (p. 268-451, 594-646)
- “VLIW Architecture for a Tree Scheduling Compiler”, R. Colwell, R. Nix, J. O’Donnell, D. Papworth, P. Rodman, ACM 1987
- Joseph Fisher - "The VLIW Machine: A Multiprocessor for Compiling Scientific Code", Computer July 1984.
- “The IBM 360/91: Machine Philosophy and Instruction Handling”, R.M. Tomasulo et al, IBM Journal of Research and Development 11:1, 1967
- “Tuning the Pentium Pro Micro-Architecture”, David Papworth, IEEE Micro, April 1996
- IA-64 Application Architecture Tutorial, Allan D. Knies, Hot-Chips 11, August 1999.

Parallelism Evolution

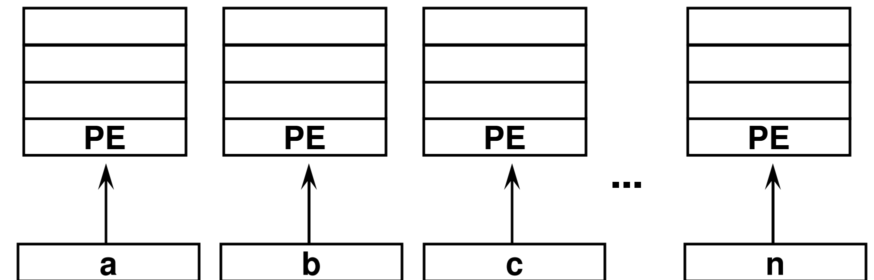
Basic configuration



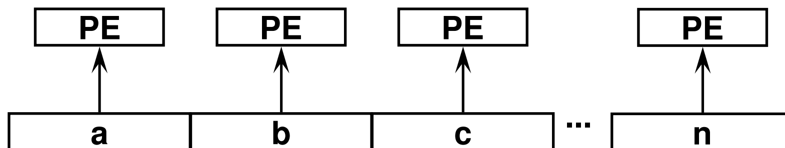
Pipeline



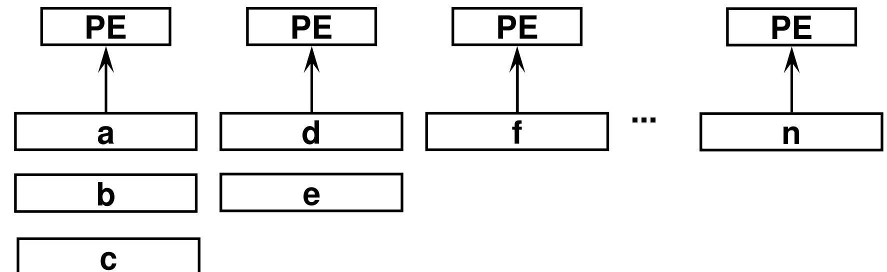
Superscalar - In order



VLIW

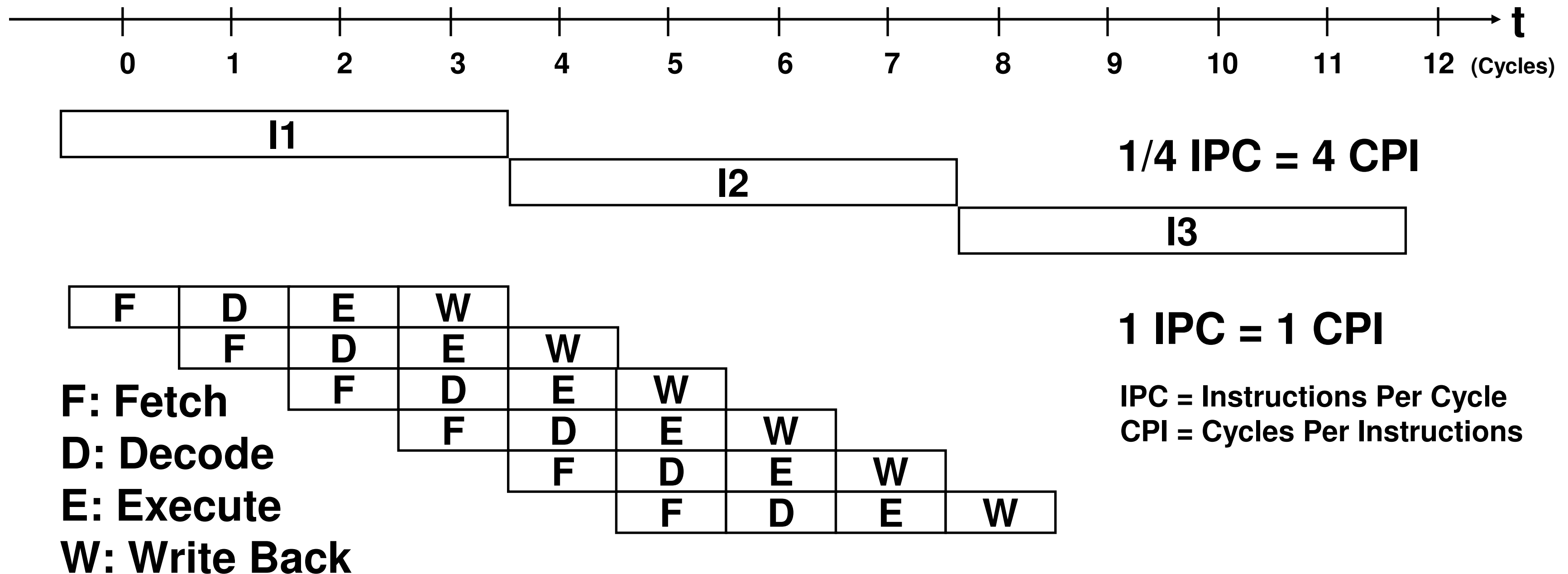


Superscalar - Out of Order



Pipeline

- Break the work to smaller pieces

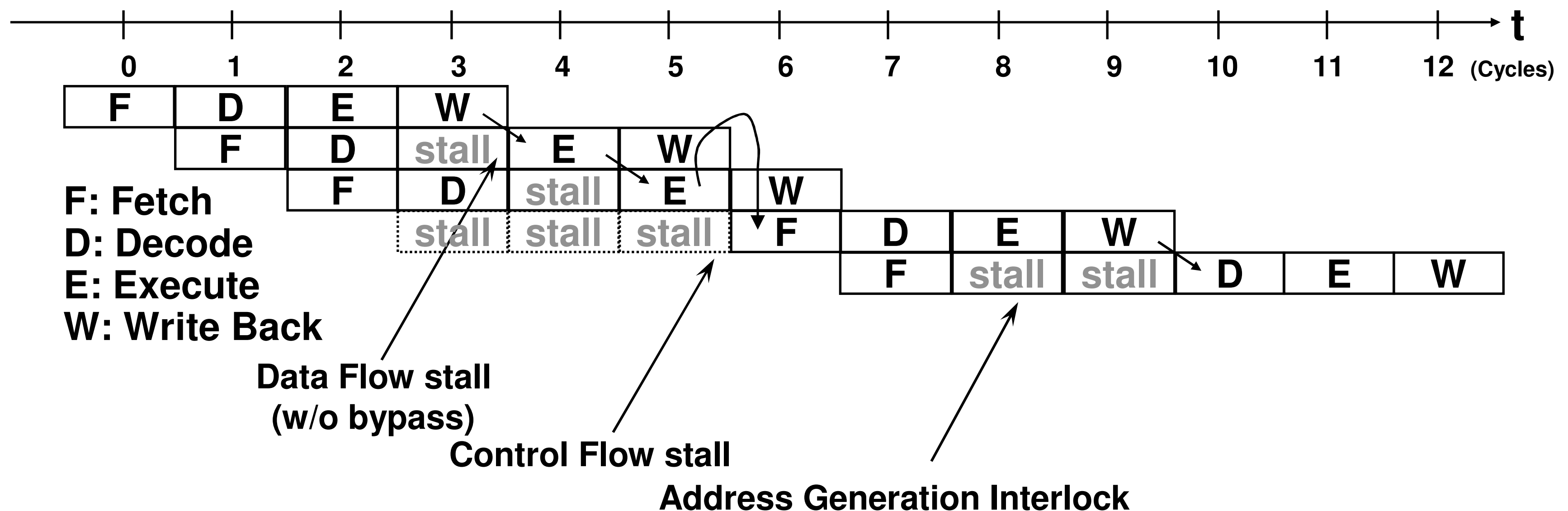


- Increased throughput
  - increased # of completed instructions per cycle and reduces cycle time
  - Number of stages varies
    - Small: 4-5 (Pentium), “Superpipeline” ~14 (Pentium Pro), “ultra-pipeline” ~25 (PIV)
- Calls for good balancing among stages

Examples
Intel 486
NS 32532

Pipeline Stalls

- But there are "stalls" in the pipeline
  - "Data Hazards": Data flow dependency (instructions output/input)
    - » Solved by: bypasses, renaming
  - "Control Hazards": Control flow dependencies
    - » Solved by branch prediction
  - "Structural Hazards": Limited resources
  - Other (Cache misses, long latency instructions, page faults….)

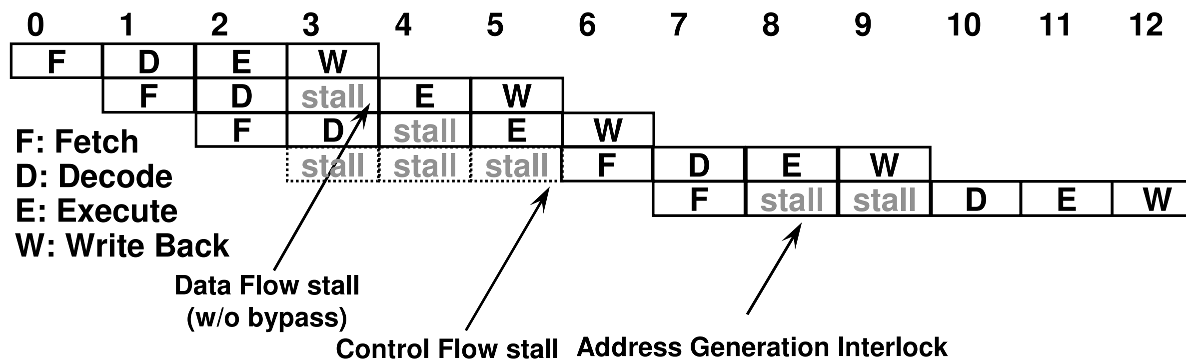


Bypasses

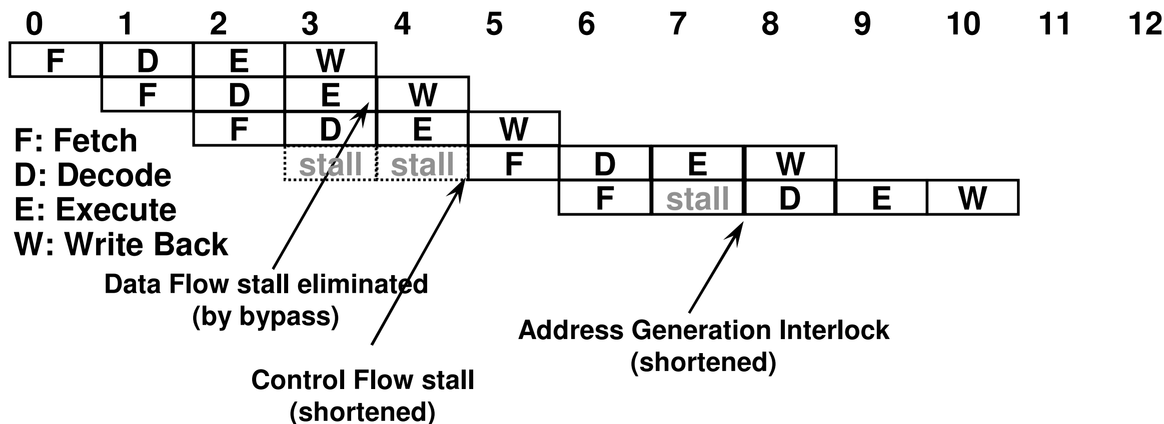
Basic mechanism to eliminate/shorten data dependencies

- Results are forwarded to their consumer before/in parallel to write back.
- Logic decides ahead of time if registers or bypasses should be used.
- There can be several level of bypass - depending on producer/consumer distance.

Pipeline w/o Bypasses

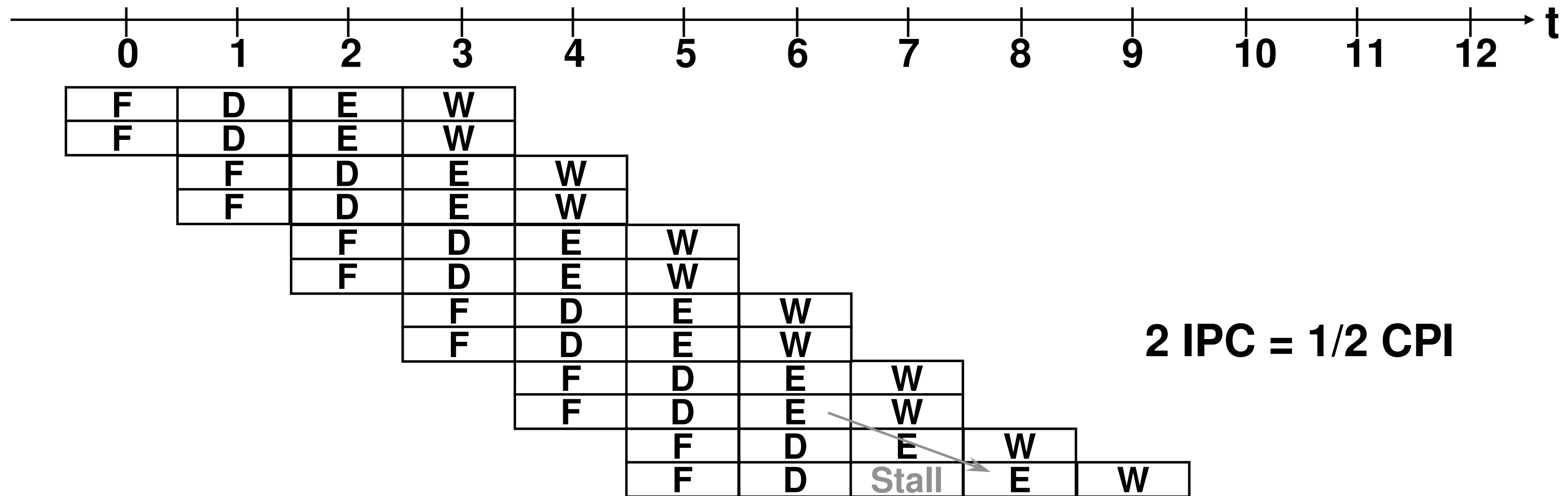


Pipeline with Bypasses



Super Scalar

- **Performs more in a single cycle**



- **Ideally, can multiply the throughput**
  - **But stall occurs more frequently**

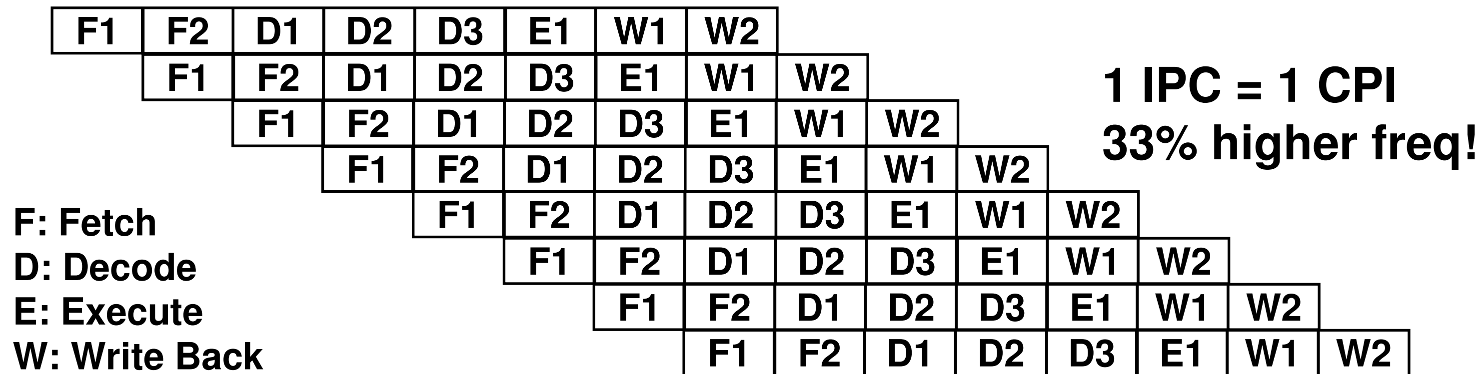
Examples
Intel Pentium® Proc.
Alpha 21164

Super Pipeline

- Split to shorter stages - allows higher frequency

Old clk = 0 1 2 3 4 5 6 7 8 9 10 11 12

New clk = 0 1 2 3 4 5 6 7 8 9 10 11 12



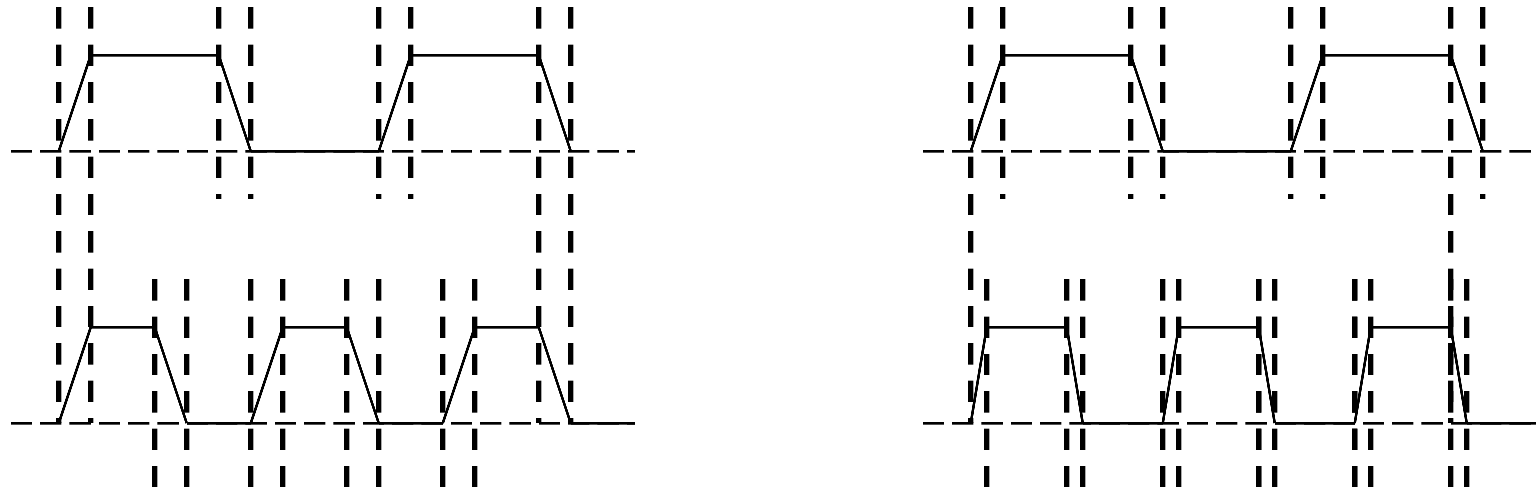
- Ideally, can (again) multiply the throughput, but
  - Stall penalties do not scale (e.g., control flow stall, cache misses)
  - Clock setup/hold reduces net cycle time - each instruction takes longer!
  - ⇨ In the example above: 2X stages, but performance gain is <33%

Examples:
Intel Pentium® II/III/4

Super Pipeline (cont...)

- **The effective clock cycle is reduced**
  - Rise and Fall times of the clock become significant
  - Latch delay take a larger portion of the cycle
  - Jitter requires relatively larger margins in design

Gaining frequency via Superpipelining vs. via Process and/or circuit techniques



Static Scheduling: VLIW / EPIC

- Static scheduling of instructions by the compiler
  - VLIW: Very Long Instruction Word (MultiFlow, TI6X family)
  - EPIC: Explicit Parallel Instruction set Computer (IA64)

☺ Shorter pipe, wider machine, global view
=> potentially huge ILP (wider & simpler than plain superscalar!)

☹ Many nops, sensitive to varying latencies (memory accesses)

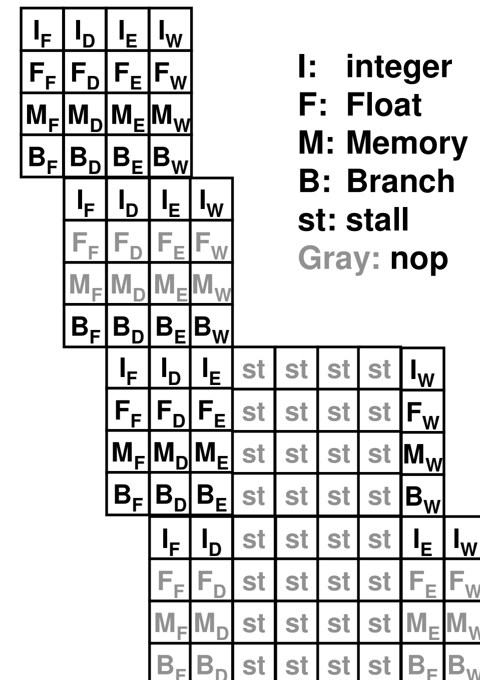
⇨ Low utilization

⇨ Huge code size

⇨ Highly depends on compiler

- EPIC overcomes some of these limitations:
  - Advance loads (hide memory latency)
  - Predicated execution (avoid branches)
  - Decoder templates (reduce nops)

  But at increased complexity.



Examples
Intel Itanium® proc.
DSPs

Dynamic Scheduling

- Scheduling instructions at run time, by the HW
- Advantages:
  - Works on the dynamic instruction flow:
    Can schedule across procedures, modules...
  - Can see dynamic values (memory addresses)
  - Can accommodate varying latencies and cases (e.g. cache miss)
- Disadvantages
  - Can schedule within a limited window only
  - Should be fast - cannot be too smart

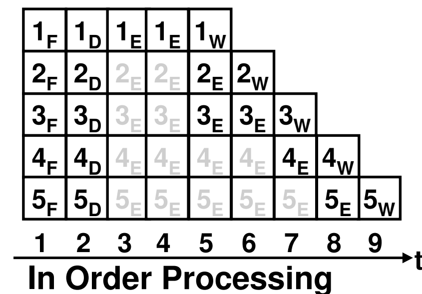
Out Of Order Execution

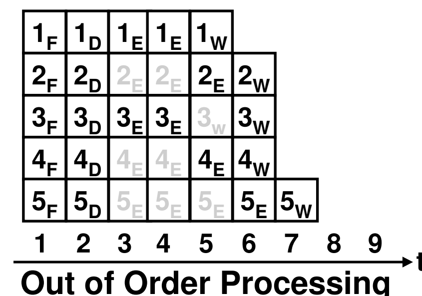
- **In Order Execution: instructions are processed in their program order.**
  - **Limitation to potential Parallelism.**
- **OOO: Instructions are executed based on "*data flow*" rather than program order**

```
Before:     src -> dest
(1) load    (r10), r21
(2) mov     r21, r31       (2 depends on 1)
(3) load    a, r11
(4) mov     r11, r22       (4 depends on 3)
(5) mov     r22, r23       (5 depends on 4)

After:
(1) load    (r10), r21;    (3) load  a, r11;
      <wait for loads to complete>
(2) mov     r21,r31;       (4) mov   r11,r22;
                           (5) mov   r22,r23;
```

- **Usually highly superscalar**





Examples:
Intel Pentium® II/III/4
Compaq Alpha 21264

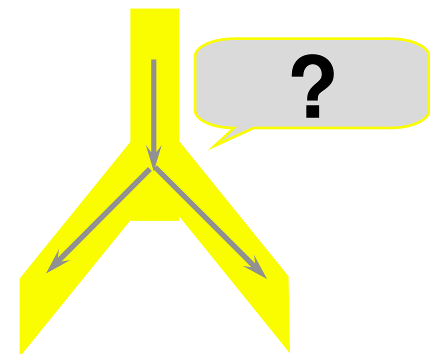
In Order vs. OOO execution.
Assuming:
- Unlimited resources
- 2 cycles load latency

Out Of Order (cont.)

- **Advantages**
  - Help exploit *Instruction Level Parallelism* (ILP)
  - Help cover latencies (e.g., cache miss, divide)
  - Artificially increase the Register file size (i.e. number of registers)
  - Superior/complementary to compiler scheduler
    - » Dynamic instruction window
    - » Make usage of more registers than the Architecture Registers
- **Complex microarchitecture**
  - Complex scheduler. Involves also
    - » Large instruction window
    - » Speculative execution
  - Requires reordering back-end mechanism (*retirement*) for:
    - » Precise interrupt resolution
    - » Misprediction/speculation recovery
    - » Memory ordering

Branch Prediction

- Goal - ensure enough instruction supply by correct prefetching
- In the past - prefetcher assumed *fall-through*
  - Lose on unconditional branch (e.g., call)
  - Lose on frequently taken branches (e.g., loops)
- Branch prediction
  - Predicts whether a branch is *taken/not taken*
  - Predicts the branch target address
- Misprediction cost varies (higher w/ increased pipeline length)
- Typical Branch prediction rates: ~90%-96%
  ➔ 4%-10% misprediction,
  ➔ 10-25 branches between mispredictions
  ➔ 50-125 instructions between mispredictions
- Misprediction cost increased with
  - Pipeline depth
  - Machine width
    » e.g. 3 width x 10 stages = 30 inst flushed!



Speculative Execution

- Execution of instructions from a predicted (yet unsure) path Eventually, path may turn wrong.
- Advantages:
  - Ensure instruction supply
  - Allow large scheduling window (for out of order)
- Issues:
  - Misprediction cost
  - Misprediction recovery

Cache - Motivation & Principle

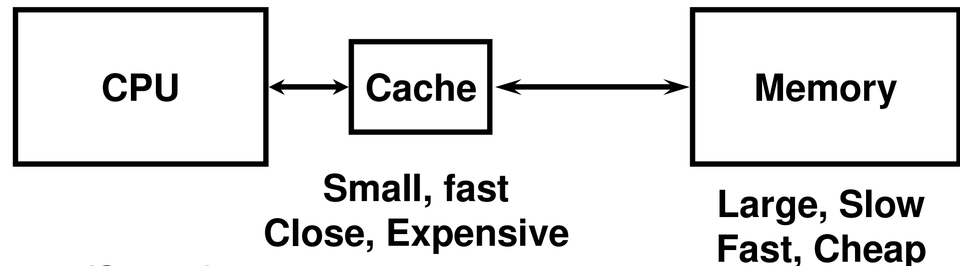
- **Memory consumption is growing about 2X every 2 years**
  - Typical size: (Y2000) 64M-128M, (Y2002) 128M-256M
- **CPU speed grows faster than memory and buses**
  - CPU/Bus grew from 1:1 to 6:1, and still growing

486	Pentium	P-II	P-III	P4
25-66MHz	66-233MHz	200-450MH	0.5-1.33GHz	1.4-2.4GHz
33MHz	66MHZ	66-100MHz	133-200MHz	400MHz

  - Memory: DRAM: 60-100ns ("10-16MHz"), Cost: <10$ per 1M
    SRAM is faster but much more expensive

  ⇨ ***Memory becomes the bottleneck for both instructions and data!***
  Slow ***or expensive***

- **Solution: Cache - A *Small, Fast, Close* memory**
  - Serves as a buffer between CPU and main memory
- **Contains copy of a portion of the main memory**
  - Small in size
  - Dynamically changed



- **Exploit space and time locality:**
  - Code is fetched sequentially (Space)
  - Code is re-executed (loops, procedures) (Time)
  - Access close or previous data (Space, Time)

Power & Performance: Qualitative look - 1

- **Energy in processing an instruction: W_i**
  - increases with the complexity of the processor.
    E.g., OOO processor consumes more energy per instruction than an in-order processor.
- **Ratio of useful to total number of processed instructions: η**
  - Total IPC including speculated instructions: IPC/η.
- **Energy/second (=power) proportional to: (IPC/η)*Frequency*W_i**
  - Proportional to the amount of processed instructions per second and the amount of work consumed per instruction.
- **Energy efficiency, measured in MIPS/Watt, proportional to: η/W_i.**
  - This value deteriorates as speculation increases and complexity grows.
- **One of the main goals of microarchitecture:**
  - Design a processor to accomplish a group of tasks, with minimum power, in the shortest amount of time, and with least amount of cost.
  - The design process involves evaluating a slew of parameters and balancing the three targets optimally with given process and circuit technology.

Power & Performance: Qualitative look - 2

- Performance
  - Perf = IPC * f
- Voltage Scaling
  - Increased operating voltage to increase frequency
  - F = k * V (within a given voltage range)
- Power & Energy consumption
  - $P = \alpha * C * V^2 * F \rightarrow P \sim \alpha * C * V^3$
  - E = P * t
- Tradeoff – depends on goal
  - Maximum performance ➔ 1% perf = a
  - Minimum energy ➔ 1% perf ≅ 1% power < W/o voltage scaling>
  - Maximum performance within constrained power ➔ 1% perf ≅ 3% power <with voltage scaling>

Obstacles for ideal improvements

- Too many things do not scale:
  - Wire delays
  - Power
  - Memory latencies and bandwidth
  - Instruction Level parallelism (ILP)

  … We solve one: we fertilize others!

➔ Machines are inherently less efficient
  - More power per useful instruction
  - Less performance per area, power, cycle
- Performance = frequency * IPC
  - Increasing IPC => more work per instruction
    - » Prediction, renaming, scheduling, etc…
  - More useless work: Speculation, replays...
  - More Frequency => More pipe stages
    - » Less gate delays per stage
    - » More gate delays per instruction overall
    - ➔ Bigger loss due to flushes, cache misses

We gain Performance => But with a lot of area and power!

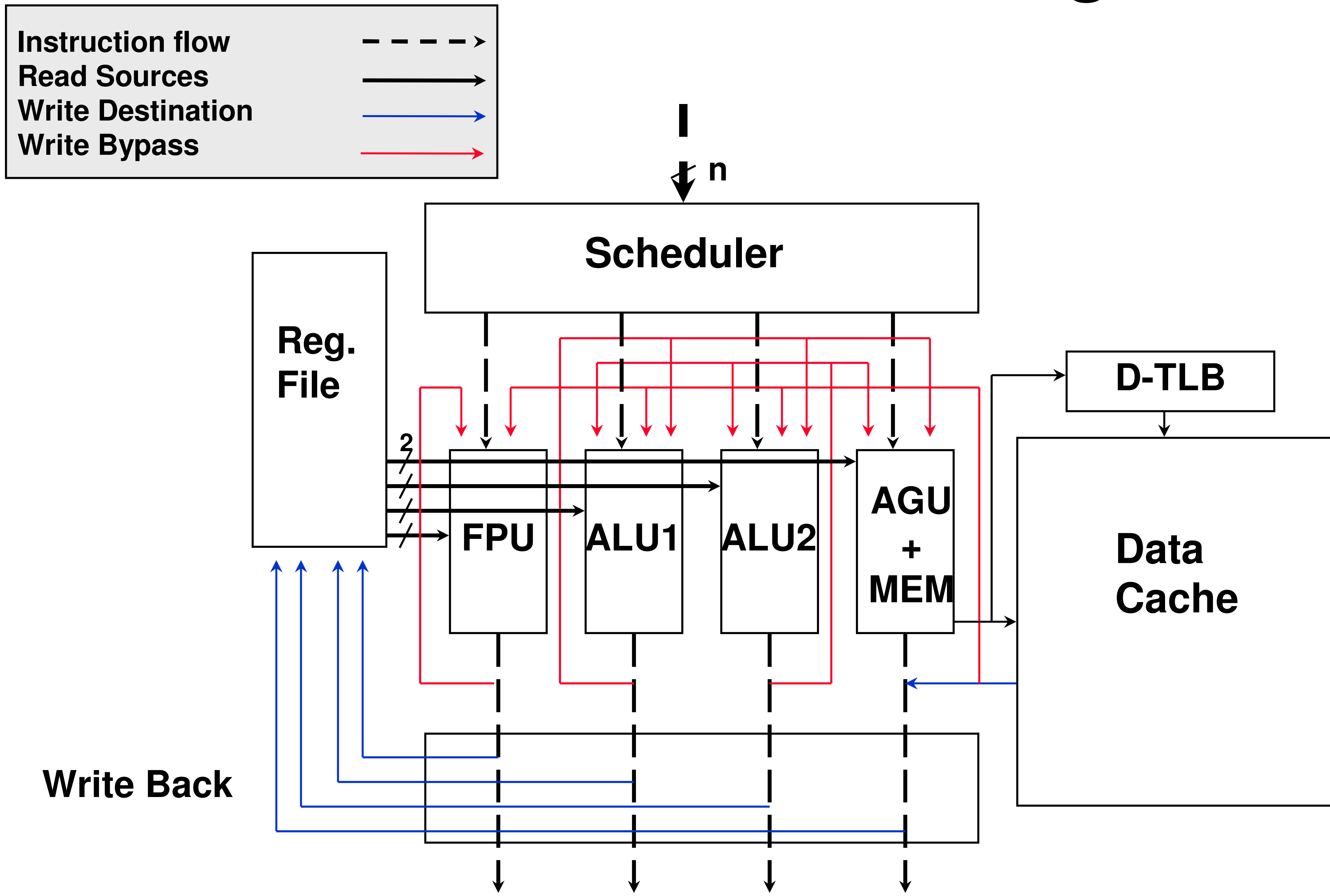
Micro-Processor Core

- Microarchitecture in a glance (“Mamas”)
  - Basic terminology and players
  - “New” vs. “Old”
  - Microarchitecture evolution
- **Micro-Processor Core**
  - **Overview**
  - **Scheduling**
- VLIW / EPIC
- Out-Of-Order

Core Structure

- Scheduling Logic
  - Static Scheduling (by the compiler - VLIW)
  - Dynamic Scheduling (by the HW – In order, Out-Of-Order)
- Register File
  - Read Sources
  - Write Destinations
- Execution Units (N)
  - Performance increase
  - Complexity increases
- Bypasses
  - Timing
  - Routing ==> Area
- Memory hierarchy
  - Translation Lookaside Buffer
  - Level-1 data cache

“The Core” - A Block Diagram



Scheduling (Issue, Dispatch) Logic

- Static Scheduling
  - Software/Compiler prepares instructions in Long Word
  - Long Word instructions are issued at the same time
- Dynamic Scheduling
  - Hardware dynamically decides which instructions are issued at the same time
- Terminology (in OOO Processor)
  - Issue: instruction grouping following the decoder
  - Dispatch: select instruction for execution

  No real distinction in In-Order processor

Issue Logic (cont’)

- Grouping of instructions into sets:
  Long Words or parallel issued set
  - Can cause “new” dependencies

e.g: Single Pipe

```
MOV  Offset(R3), R4
ADD   R4, R1
MOV  Offset(R4), R5
ADD   R5, R4
```

e.g: Dual Pipe

```
MOV Offset(R3), R4      ADD  R1, R4  // should recognize dependencies
MOV Offset(R4), R5      ADD  R5, R4
```

e.g: Dual Pipe (in execution)

```
MOV  Offset(R3), R4                  // dependencies OK
ADD   R4, R1           MOV  Offset(R4), R5
ADD   R5, R4
```

- Grouping of instructions into sets, reduces the distance between branches

Scheduling In-Order Superscalar RISC

- **In-order scheduler Logic assuming N-way superscalar**
  - **Predict control flow**
  - **Look at the next N consecutive instructions to locate candidates for parallel dispatch**
  - **Detect resource collisions**
  - **Detect data dependence**
  - **Check for Source-Ready accounting for all bypasses**

  - **Dispatch until the first conflicting instruction**

Micro-Processor Core

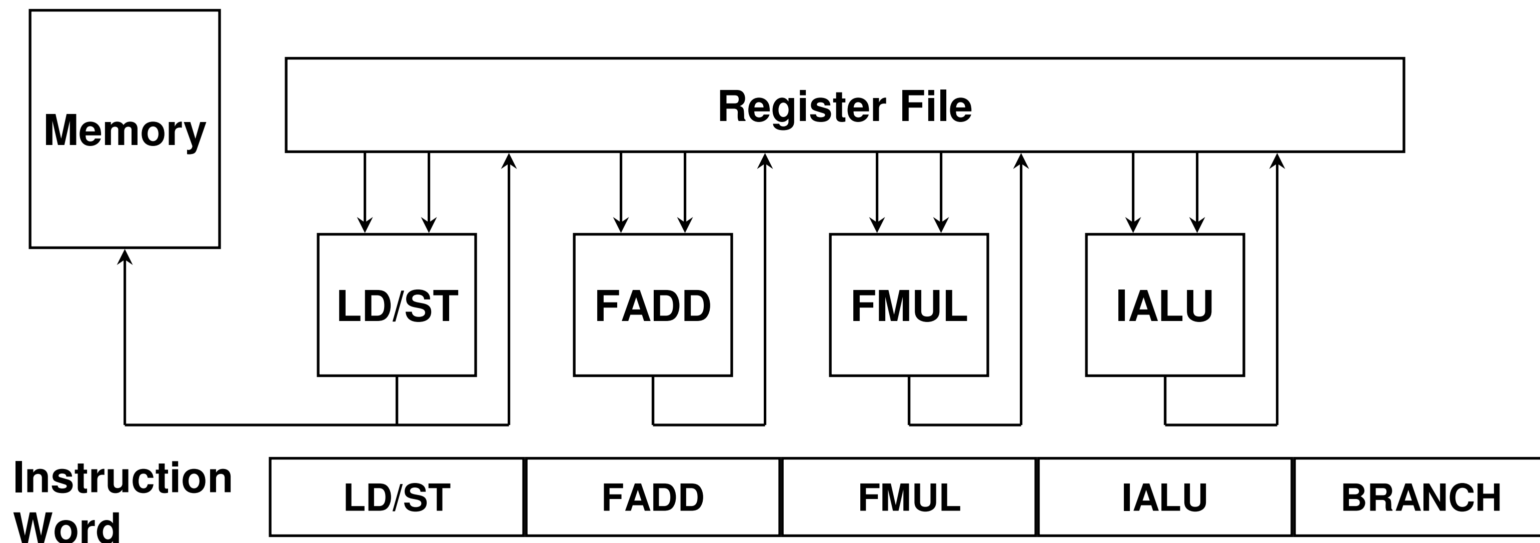
- Microarchitecture in a glance (“Mamas”)
  - Basic terminology and players
  - “New” vs “Old”
  - Microarchitecture evolution
- Micro-Processor Core
  - Overview
  - Scheduling
- **VLIW / EPIC**
- Out-Of-Order

Contribution by:
Uri Weiser, Intel

Static Issuing - example
VLIW-Very Long Instruction Word

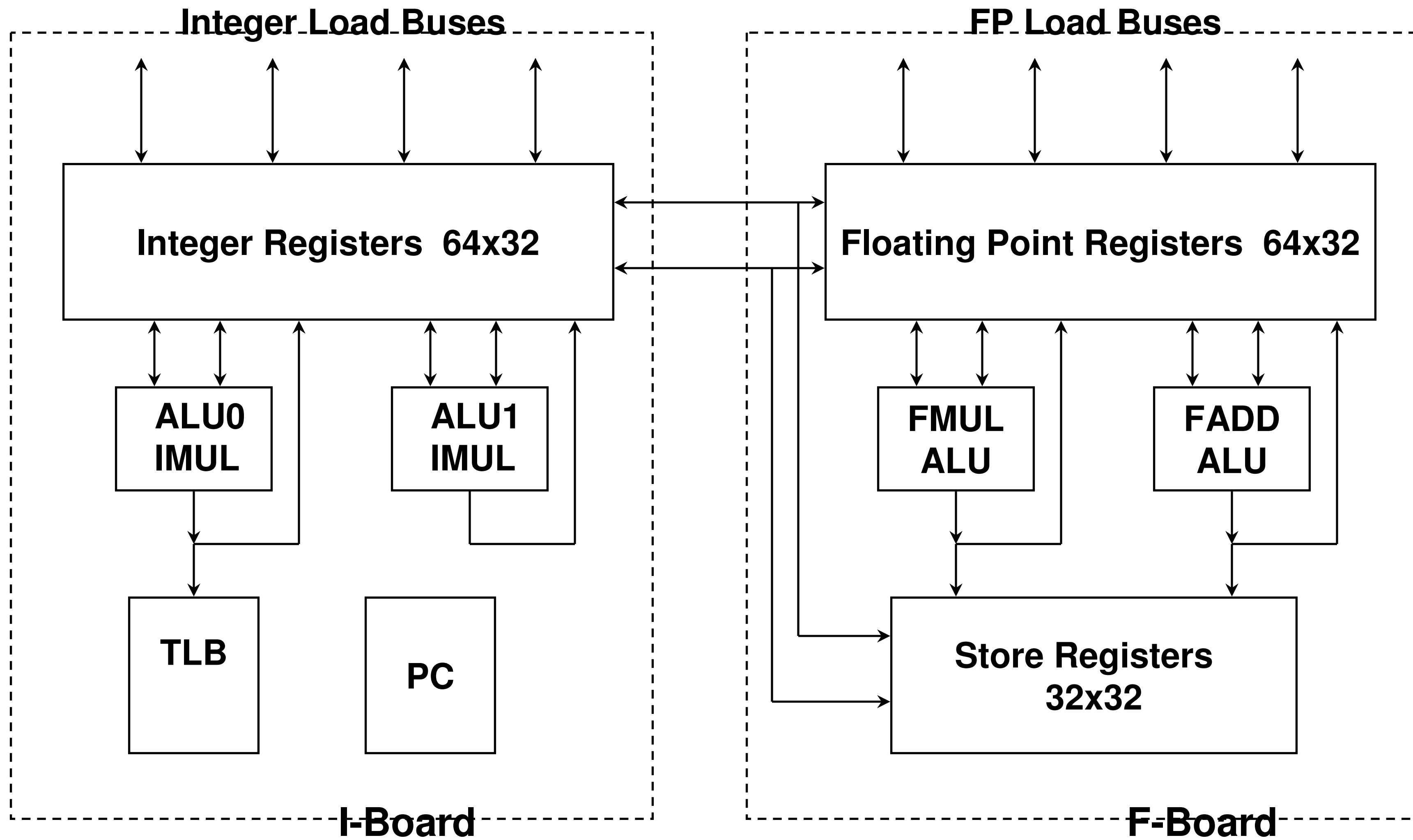
Multiflow 7/200

- **A VLIW Performs many program steps at once.**
- **Many operations are grouped together into Very Long Instruction Word and execute together**



Ref: “VLIW Architecture for a Trace Scheduling Compiler” Colwell. Nix, O’Donnell

Multiflow 7/200 (cont')



Multiflow 7/200 (cont')

Multiflow Trace 7/200 Machine is constructed of Integer-FP pairs:

Integer board: 64 32-bit registers
Full ALU (including multiplier) W/ Access to Reg file
Full ALU (including multiplier) W/ Access to both Reg File and Memory*
TLB
Program Counter

FP board: 64 32-bit registers
FP mult w/ simple ALU (no multiplier)
FP adder w/ simple ALU (no multiplier)
Store Register File - 32 32-bit registers

(n) execution time in Beats
* Memory access throughput 1 Beat
latency 7 Beats

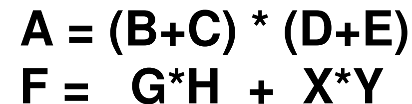
Multiflow 7/200 (cont’)

Compiler Basic Concept

Optimized compiler arrange instructions according to instruction timing

example:

```
LD      #B, R1
LD      #C, R2
FADD    R1, R2, R3
LD      #D, R4
LD      #E, R5
FADD    R4, R5, R6
FMUL    R6, R3, R1
STO     R1, #A
LD      #G, R7
LD      #H, R8
FMULL   R7, R8, R9
LD      #X, R4
LD      #Y, R5
FMULL   R4, R5, R6
FADD    R6, R9, R1
STO     R1, #F
```



Assume latencies:

Load	3
FADD	3
FMUL	3
Store	1

Multiflow 7/200 (cont')

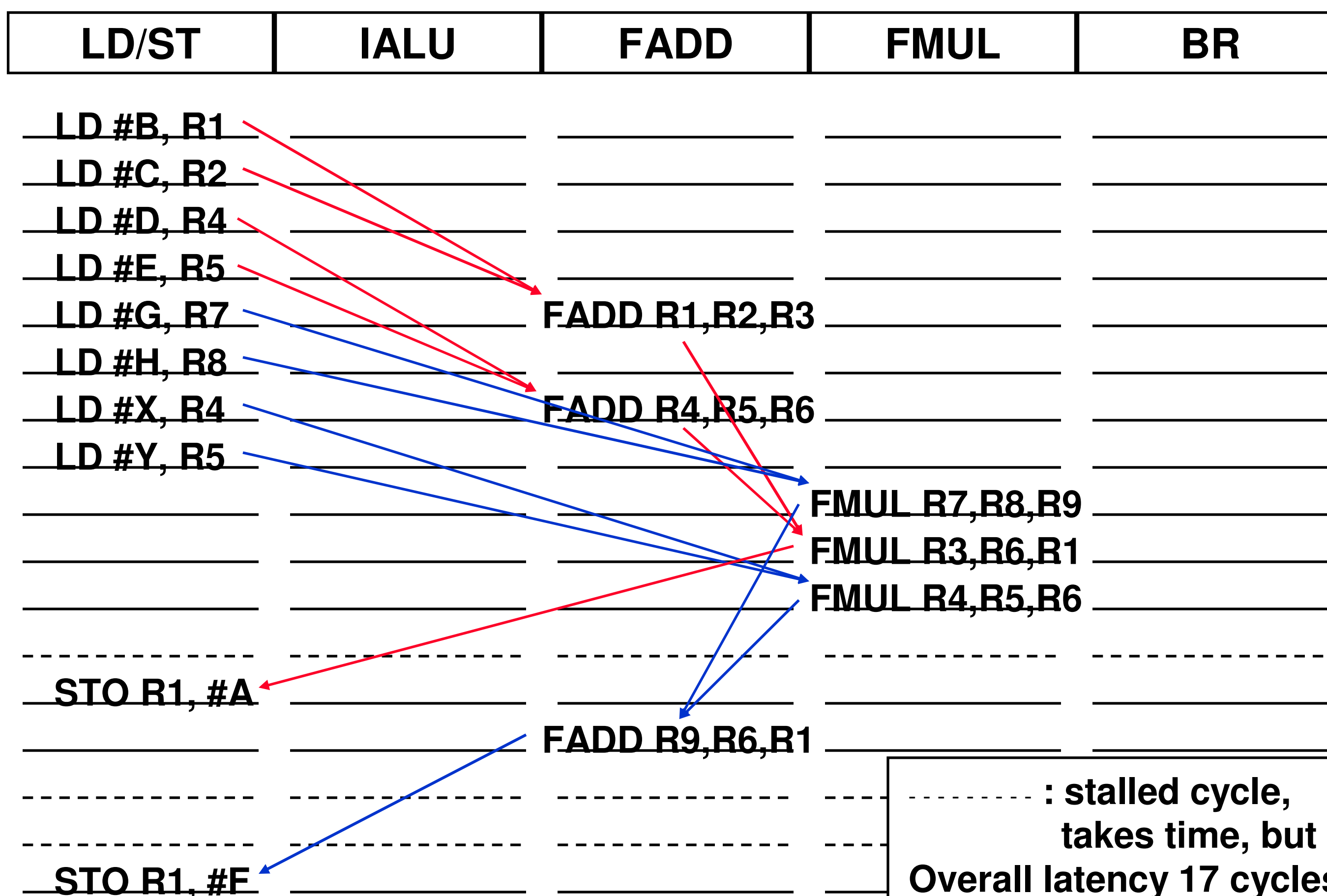
Compiler Basic Concept

Assume latencies:

Load	3
FADD	3
FMUL	3
Store	1

Example (Cont.): A = (B+C) * (D+E)

F = G*H + X*Y

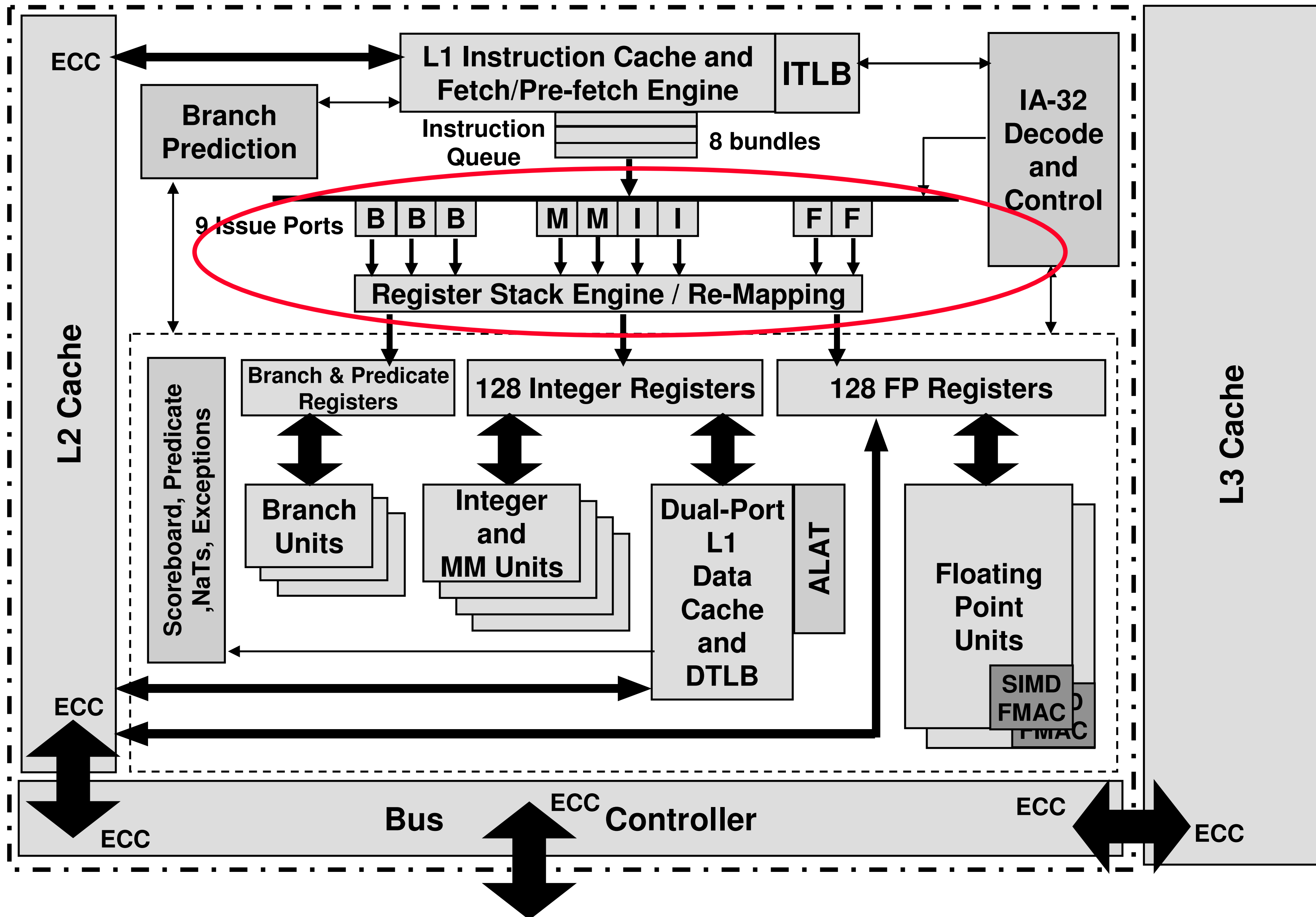


- - - - - : stalled cycle, takes time, but no space.

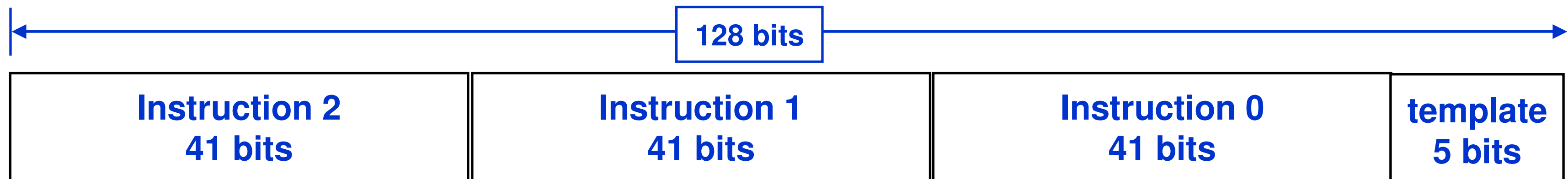
Overall latency 17 cycles.

Very Low code efficiency: <25%!

Intel® Itanium™ Processor Block Diagram



IA64 Instruction Template



- Instruction Types
  - M: Memory
  - I: Shifts, MM
  - A: ALU
  - B: Branch
  - F: Floating point
  - L+X: Long
- Template types
  - Regular: MII, MLX, MMI, MFI, MMF
  - Stop: MI_I M_MI
  - Branch: MIB, MMB, MFB, MBB, BBB
  - All come in two versions:
    - » with stop at end
    - » without stop at end

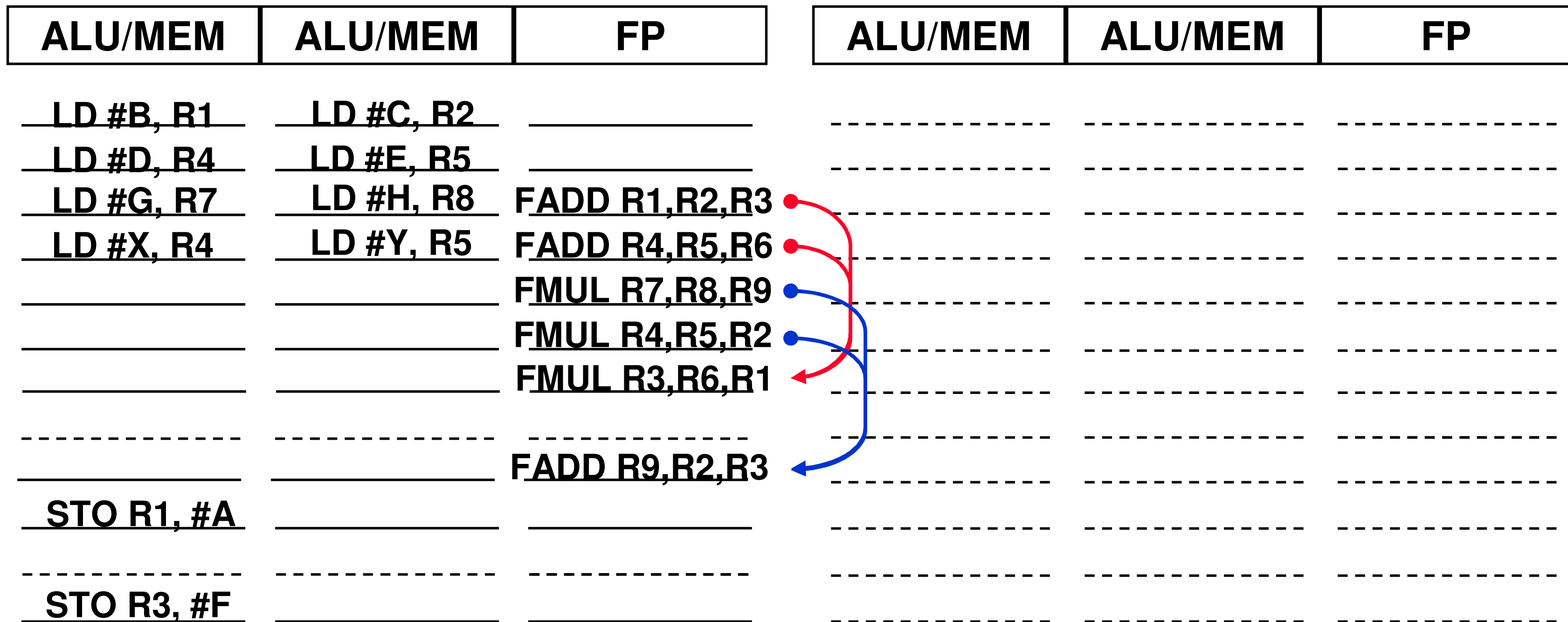
- Microarchitecture considerations:
  - Can run N bundles per clock (Merced = 2)
  - Limits on numbers of memory ports (Merced =2, future > 2?)

IA64 - Itanium™ Processor
Compiler Basic Concept

Assume latencies:

Load	2 (2 ports)
FADD	3
FMUL	3
Store	1

Example (Cont.): A = (B+C) * (D+E)
F = G*H + X*Y



- In VLIW/EPIC compiler should know μarch very well
  - To exploit performance, code should be recompiled if latencies are changed.
- In IA64:
  - More versatile instruction grouping
  - Dependent instructions can reside in the same word, to save code space.

Overall latency 12 cycles.
Code Efficiency ~55%!

IA64 - "Post-Itanium" (cont')

Compiler Basic Concept

Assume latencies:

Load	2 (4 ports)
FADD	3
FMUL	3
Store	1

Example (Cont.): A = (B+C) * (D+E)
F = G*H + X*Y

ALU/MEM	ALU/MEM	FP	ALU/MEM	ALU/MEM	FP
LD #B, R1	LD #C, R2		LD #D, R4	LD #E, R5	
LD #G, R7	LD #H, R8		LD #X, R9	LD #Y, R10	
		FADD R1,R2,R3			FADD R4,R5,R6
		FMUL R7,R8,R1			FMUL R9,R10,R2
		FMUL R3,R6,R4			
		FADD R1,R2,R5			
STO R4, #A					
STO R5, #F					

- **Microarchitecture changes should affect generated code:**
  - e.g., 1st FADDs cannot be combined w/ loads as in previous generation

Overall latency 10 cycles.
Code Efficiency ~45%!