# POLITECNICO DI MILANO

μ-LAB

High Performance Processors and Systems

# *Instruction Level Parallelism*

## VLIW architectures

**Donatella Sciuto: sciuto@elet.polimi.it**

**Marco D. Santambrogio: marco.santambrogio@polimi.it**

HPPS

# Outline

- Introduction to ILP
- Dependences and hazards
- Static scheduling
- VLIW architecture

# Definition of ILP

- **ILP = Potential overlap of execution among unrelated instructions**

- Overlapping possible if:
  - No Structural Hazards
  - No RAW, WAR of WAW Stalls
  - No Control Stalls

# Review: Pipeline performance

- Pipeline CPI = Ideal pipeline CPI + Structural Stalls + Data Hazard Stalls + Control Stalls
  - Ideal pipeline CPI: measure of the maximum performance attainable by the implementation
  - Structural hazards: HW cannot support this combination of instructions
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches, jumps, exceptions)

# Review: Summary of Pipelining Basics

- Hazards limit performance
  - Structural: need more HW resources
  - Data: need forwarding, compiler scheduling
  - Control: early evaluation & PC, delayed branch, prediction
- Increasing length of pipe increases impact of hazards
- Pipelining helps instruction bandwidth, not latency

# Review: Summary of Pipelining Basics

- Interrupts, Instruction Set, FP makes pipelining harder
- Compilers reduce cost of data and control hazards
  - Load delay slots
  - Branch delay slots
  - Branch prediction
- Today: Longer pipelines => Better branch prediction, more instruction parallelism?

# Review: Types of Data Hazards

Consider executing a sequence of

$$r_k \leftarrow (r_i) \ \text{op} \ (r_j)$$

Type of instructions

## Data-dependence

$r_3 \leftarrow (r_1)$ op $(r_2)$ — Read-after-Write

$r_5 \leftarrow (r_3)$ op $(r_4)$ — (RAW) hazard

## Anti-dependence

$r_3 \leftarrow (r_1)$ op $(r_2)$ — Write-after-Read

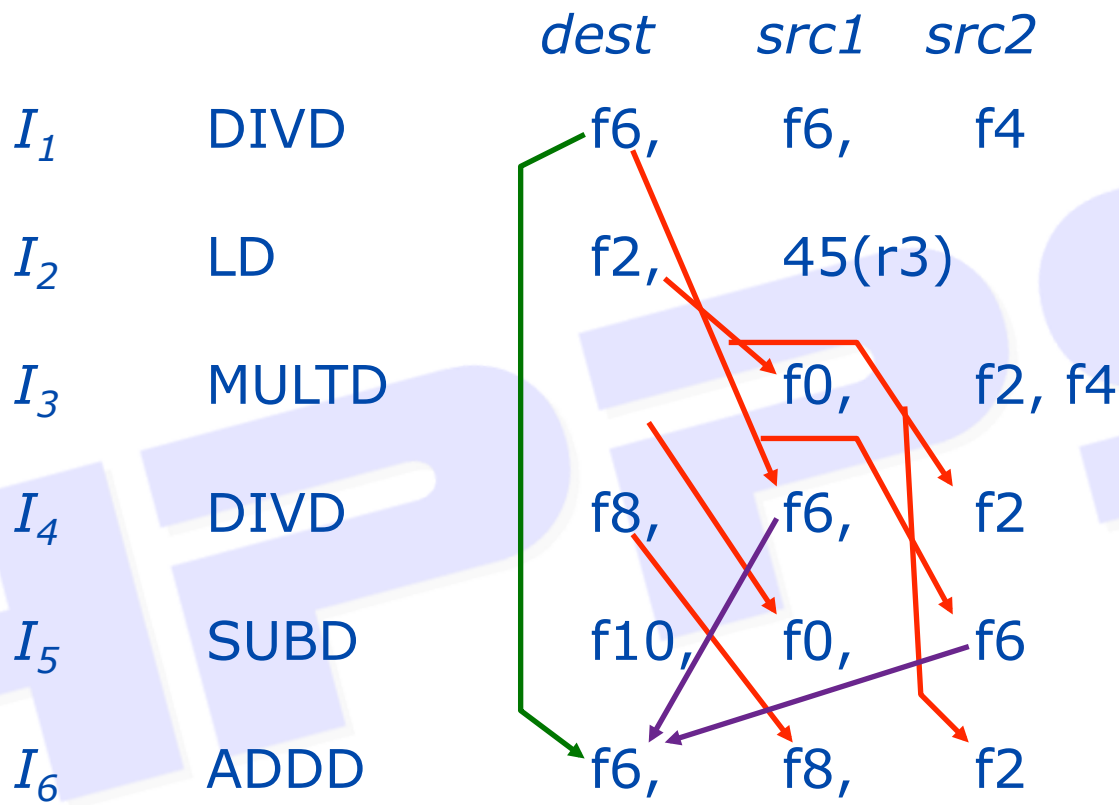
$r_1 \leftarrow (r_4)$ op $(r_5)$ — (WAR) hazard

## Output-dependence

$r_3 \leftarrow (r_1)$ op $(r_2)$ — Write-after-Write

$r_3 \leftarrow (r_6)$ op $(r_7)$ — (WAW) hazard

# Data Hazards: An Example

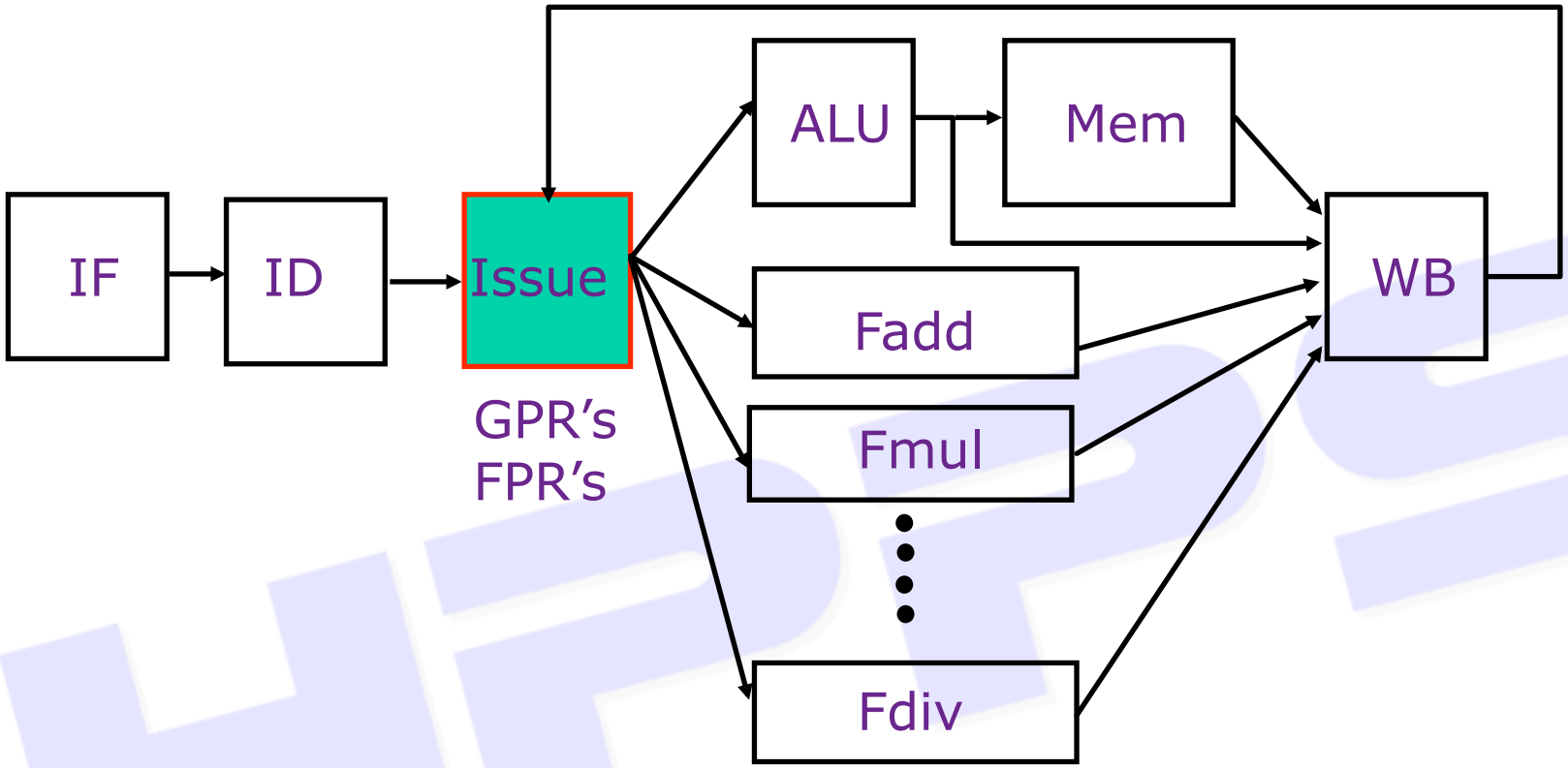



*RAW Hazards*

*WAR Hazards*

*WAW Hazards*

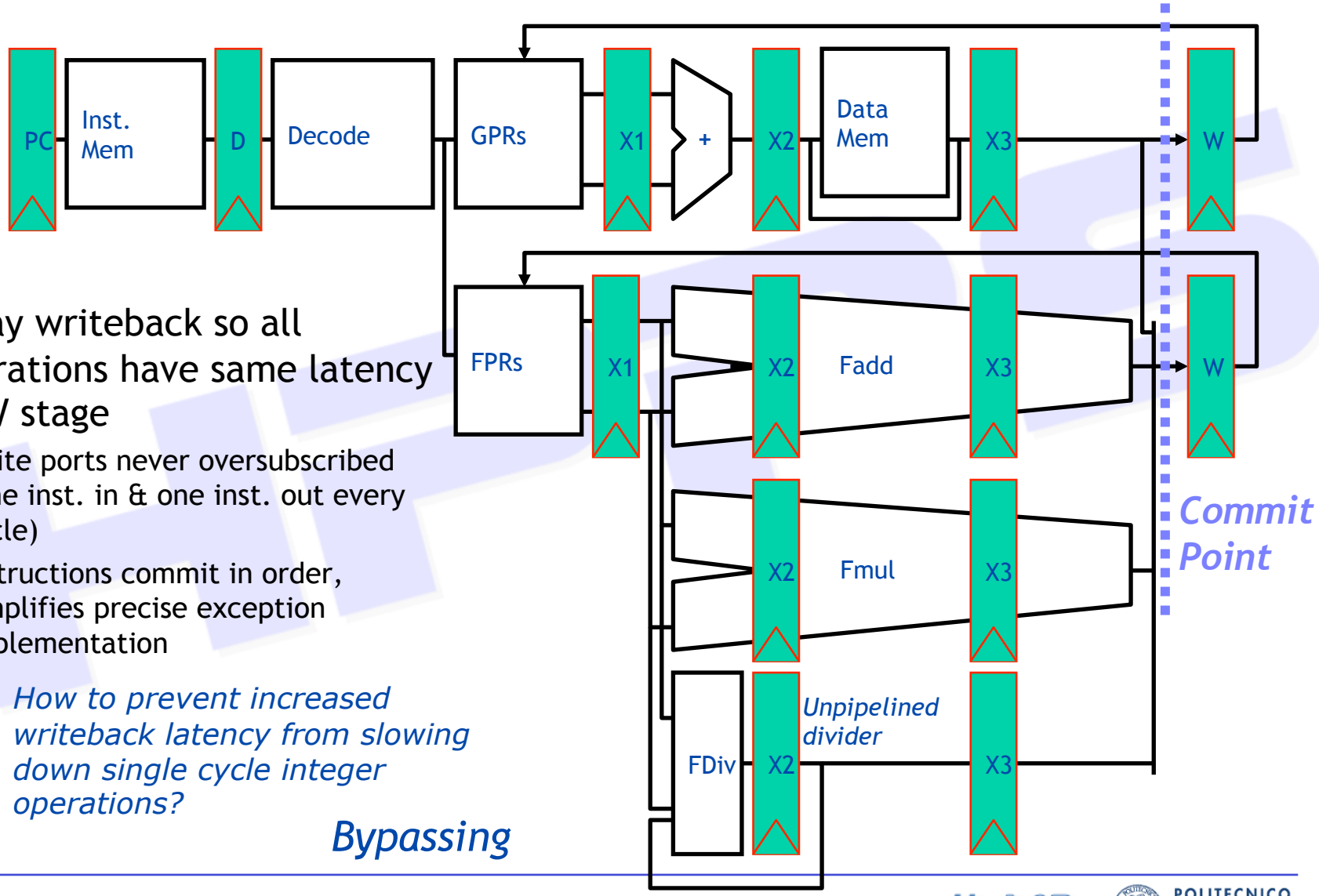
# Complex Pipelining




Pipelining becomes complex when we want high performance in the presence of:

- Long latency or partially pipelined floating-point units
- Multiple function and memory units
- Memory systems with variable access time
- Precise exception

# Complex In-Order Pipeline



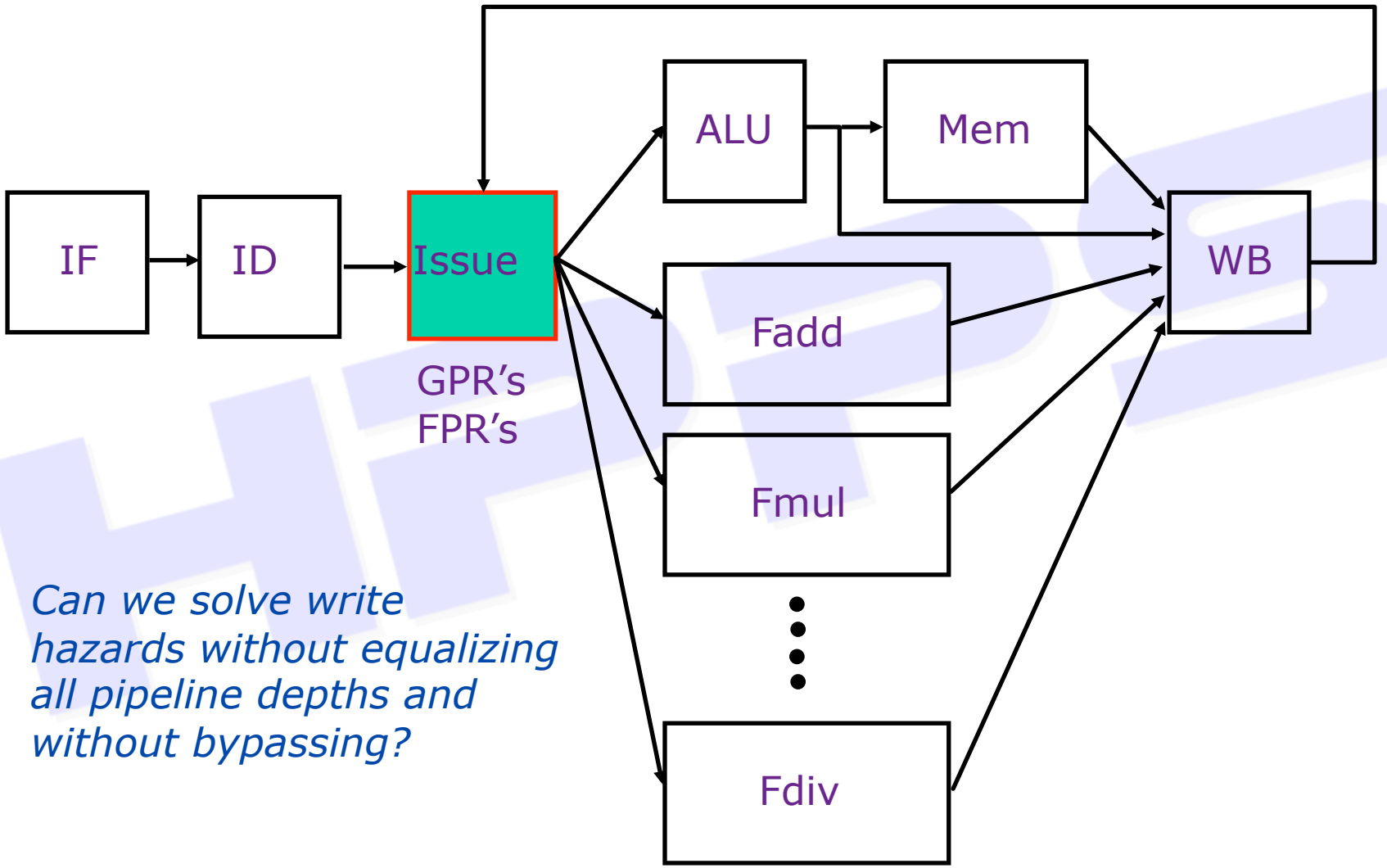

Delay writeback so all operations have same latency to W stage

- Write ports never oversubscribed (one inst. in & one inst. out every cycle)
- Instructions commit in order, simplifies precise exception implementation

*How to prevent increased writeback latency from slowing down single cycle integer operations?*

*Bypassing*

# Complex Pipeline




*Can we solve write hazards without equalizing all pipeline depths and without bypassing?*

# When is it Safe to Issue an Instruction?

- Suppose a data structure keeps track of all the instructions in all the functional units
- The following checks need to be made before the Issue stage can dispatch an instruction
  - Is the required function unit available?
  - Is the input data available? ⇒ RAW?
  - Is it safe to write the destination? ⇒ WAR? WAW?
  - Is there a structural conflict at the WB stage?

# Getting higher performance...

- In a pipelined machine, actual CPI is derived as:

  $CPI_{pipe} = CPI_{ideal}$ + Structural_stalls +

  Data_hazard_stalls + Control_stalls

- Reduction of any right-hand term reduces $CPI_{pipe}$ (increase *Instructions Per Clock - IPC*)
- Technique to increase $CPI_{pipe}$ could create further problems with hazards.

# Some basic concepts and definitions

- To reach higher performance (for a given technology) - more parallelism must be extracted from the program. In other words...
- Dependences must be detected and solved, and instructions must be *ordered* (***scheduled***) so as to achieve highest parallelism of execution compatible with available resources.

# Dependences

- Determining dependences among instructions is critical to defining the amount of parallelism existing in a program.
- If two instructions are dependent, they cannot execute in parallel: they must be executed in order or only partially overlapped.
- Three different types of dependences:
  - **Data Dependences (or True Data Dependences)**
  - **Name Dependences**
  - **Control Dependences**

# Name Dependences

- Name dependence occurs when 2 instructions use the same register or memory location (called **name**), but there is no flow of data between the instructions associated with that name.
- Two type of name dependences between an instruction **i** that precedes instruction **j** in program order:
  - **Antidependence**: when **j** writes a register or memory location that instruction **i** reads. The original instructions ordering must be preserved to ensure that **i** reads the correct value.
  - **Output Dependence**: when **i** and **j** write the same register or memory location. The original instructions ordering must be preserved to ensure that the value finally written corresponds to **j**.

# Name Dependences

- Name dependences are not true data dependences, since there is no value (no data flow) being transmitted between instructions.
- If the name (register number or memory location) used in the instructions could be changed, the instructions do not conflict.
- Dependences trough memory locations are more difficult to detect ("memory disambiguation" problem), since two addresses may refer to the same location but can look different.
- **Register renaming** can be more easily done.
- Renaming can be done either statically by the compiler or dynamically by the hardware.

# Data Dependences and Hazards

- A data/name dependence can potentially generate a data hazard (**RAW**, **WAW**, or **WAR**), but the actual hazard and the number of stalls to eliminate the hazards are a property of the pipeline.
  - **RAW** hazards correspond to true data dependences.
  - **WAW** hazards correspond to output dependences
  - **WAR** hazards correspond to antidependences.
- **Dependences are a property of the program, while hazards are a property of the pipeline.**

# Control Dependences

- Control dependence determines the ordering of instructions and it is preserved by two properties:
  - Instructions execution in program order to ensure that an instruction that occurs before a branch is executed before the branch.
  - Detection of control hazards to ensure that an instruction (that is control dependent on a branch) is not executed until the branch direction is known.
- Although preserving control dependence is a simple way to preserve program order, **control dependence is not the critical property** that must be preserved.

# Program Properties

- **Two properties** are critical to program correctness (and normally preserved by maintaining both data and control dependences):
  - **Exception behavior:** Preserving exception behavior means that any changes in the ordering of instruction execution must not change how exceptions are raised in the program.
  - **Data flow:** Actual flow of data values among instructions that produces the correct results and consumes them.

# Instruction Level Parallelism

- Two strategies to support ILP:
  - **Dynamic Scheduling:** Depend on the hardware to locate parallelism
  - **Static Scheduling:** Rely on software for identifying potential parallelism
- Hardware intensive approaches dominate desktop and server markets

# Dynamic Scheduling

- The hardware reorder the instruction execution to reduce pipeline stalls while maintaining data flow and exception behavior.
- Main advantages:
  - It enables handling some cases where dependences are unknown at compile time
  - It simplifies the compiler complexity
  - It allows compiled code to run efficiently on a different pipeline.
- Those advantages are gained at a cost of a significant increase in hardware complexity and power consumption.

# Static Scheduling

- Compilers can use sophisticated algorithms for code scheduling to exploit **ILP (Instruction Level Parallelism).**
- The amount of parallelism available within a **basic block** – a straight-line code sequence with no branches in except to the entry and no branches out except at the exit – is quite small.
- Example: For typical MIPS programs the average branch frequency is between 15% and 25% ⇒ from 4 to 7 instructions execute between a pair of branches.

# Static Scheduling

- Data dependence can further limit the amount of ILP we can exploit within a basic block to much less than the average basic block size.
- To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks (i.e. across branches).

# Detection and resolution of dependences: Static Scheduling

- Static detection and resolution of dependences (⇨ **static scheduling**): accomplished by the compiler ⇨ dependences are avoided by code reordering.
  Output of the compiler: reordered into dependency-free code.

- Typical example: VLIW (Very Long Instruction Word) processors expect ***dependency-free code***.

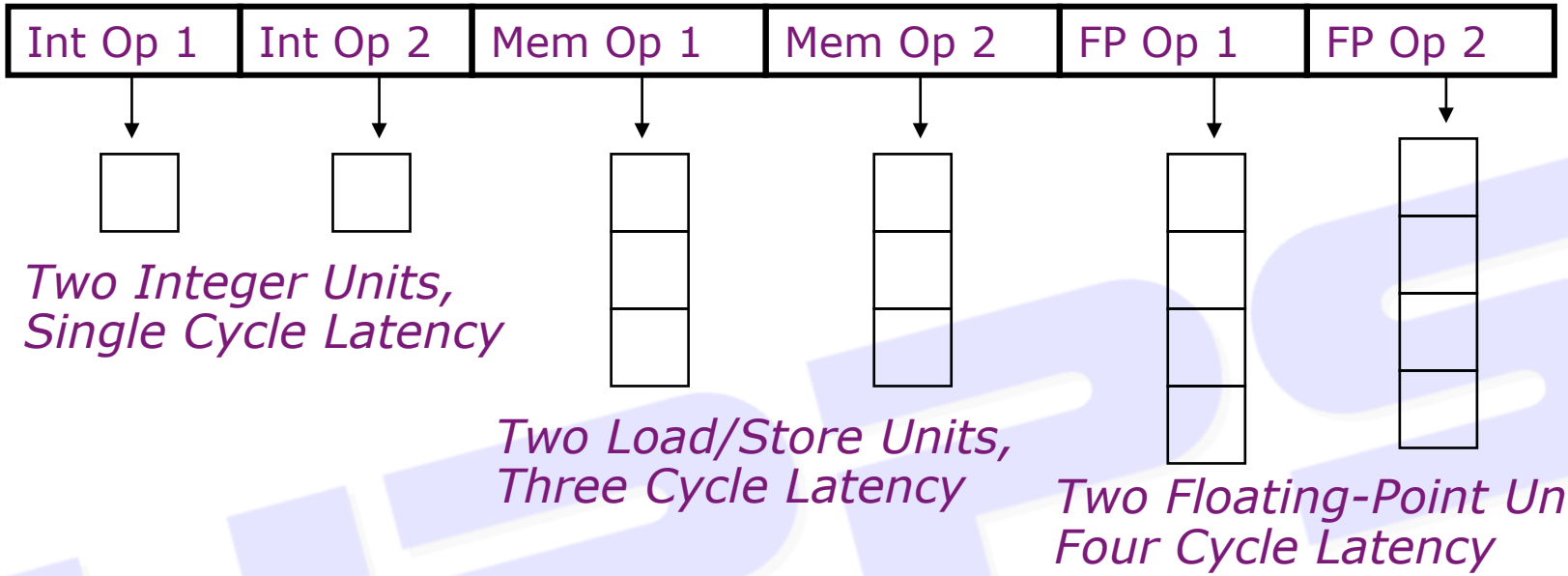
# Limits of Static Scheduling

- Unpredictable branches
- Variable memory latency (unpredictable cache misses)
- Code size explosion
- Compiler complexity

# VLIW processors

- Very Long Instruction Word Architectures
  - Oriented towards the exploitation of Instruction-level Parallelism
  - Low hardware complexity (no scheduling hardware, reduced support of variable latency instructions)
  - No instruction reordering performed by the hardware
  - Explicit parallelism
  - Single control flow

# VLIW: Very Long Instruction Word




- Multiple operations packed into one instruction
- Each operation slot is for a fixed function
- Constant operation latencies are specified
- Architecture requires guarantee of:
  - Parallelism within an instruction => no x-operation RAW check
  - No data use before data ready => no data interlocks

# VLIW Compiler Responsibilities

The compiler:

- Schedules to maximize parallel execution
  - Exploit ILP and LLP
  - It is necessary to map the instructions over the machine functional units
  - This mapping must account for time constraints and dependencies among the tasks
- Guarantees intra-instruction parallelism
- Schedules to avoid data hazards (no interlocks)
  - Typically separates operations with explicit NOPs
- The goal is to minimize the total execution time for the program

# Early VLIW Machines

- FPS AP120B (1976)
  - scientific attached array processor
  - first commercial wide instruction machine
  - hand-coded vector math libraries using software pipelining and loop unrolling
- Multiflow Trace (1987)
  - commercialization of ideas from Fisher's Yale group including "trace scheduling"
  - available in configurations with 7, 14, or 28 operations/instruction
  - 28 operations packed into a 1024-bit instruction word
- Cydrome Cydra-5 (1987)
  - 7 operations encoded in 256-bit instruction word
  - rotating register file

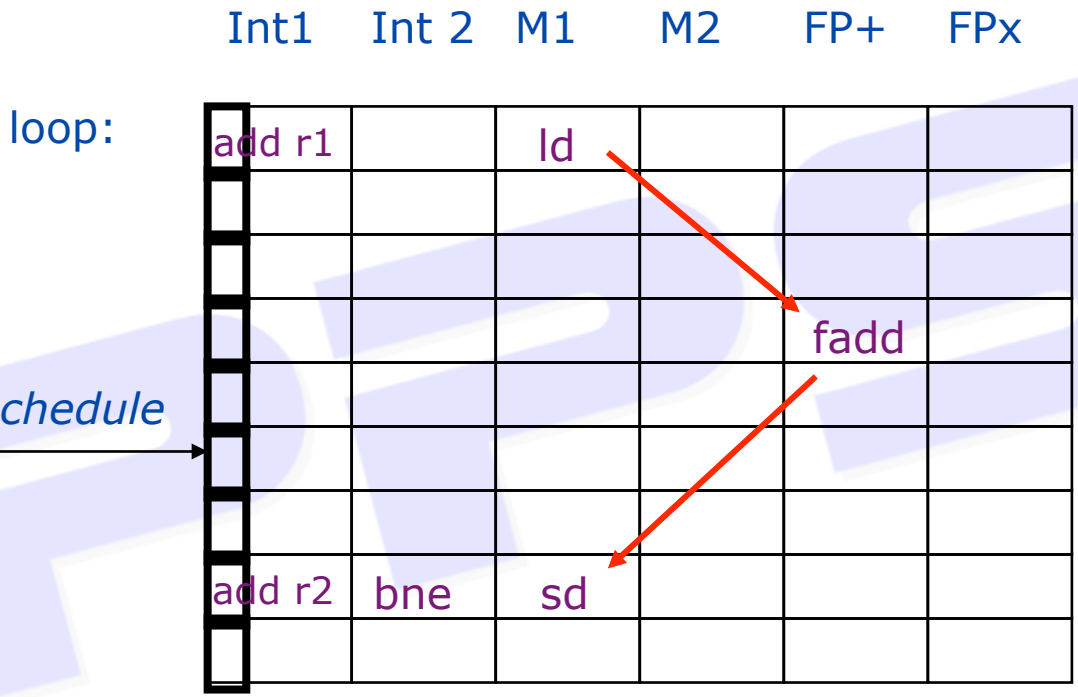
# Loop Execution

```
for (i=0; i<N; i++)
    B[i] = A[i] + C;
```

*Compile*

```
loop:  ld f1, 0(r1)
       add r1, 8
       fadd f2, f0, f1
       sd f2, 0(r2)
       add r2, 8
       bne r1, r3, loop
```

*Schedule*




How many FP ops/cycle?

1 fadd / 8 cycles = 0.125

# Loop Unrolling

```
for (i=0; i<N; i++)
    B[i] = A[i] + C;
```

Unroll inner loop to perform 4 iterations at once

```
for (i=0; i<N; i+=4)
{
    B[i]   = A[i] + C;
    B[i+1] = A[i+1] + C;
    B[i+2] = A[i+2] + C;
    B[i+3] = A[i+3] + C;
}
```

Need to handle values of N that are not multiples of unrolling factor with final cleanup loop

# Scheduling Loop Unrolled Code

*Unroll 4 ways*

```
loop:  ld f1, 0(r1)
       ld f2, 8(r1)
       ld f3, 16(r1)
       ld f4, 24(r1)
       add r1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       sd f5, 0(r2)
       sd f6, 8(r2)
       sd f7, 16(r2)
       sd f8, 24(r2)
add r2, 32
       bne r1, r3, loop
```

*Schedule* →

loop:

| Int1 | Int 2 | M1 | M2 | FP+ | FPx |
|---|---|---|---|---|---|
| | | ld f1 | | | |
| | | ld f2 | | | |
| | | ld f3 | | | |
| add r1 | | ld f4 | | fadd f5 | |
| | | | | fadd f6 | |
| | | | | fadd f7 | |
| | | | | fadd f8 | |
| | | sd f5 | | | |
| | | sd f6 | | | |
| | | sd f7 | | | |
| add r2 | bne | sd f8 | | | |
| | | | | | |
| | | | | | |

How many FLOPS/cycle?

4 fadds / 11 cycles = 0.36

# Software Pipelining

*Unroll 4 ways first*

```
loop:  ld f1, 0(r1)
       ld f2, 8(r1)
       ld f3, 16(r1)
       ld f4, 24(r1)
       add r1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       sd f5, 0(r2)
       sd f6, 8(r2)
       sd f7, 16(r2)
       add r2, 32
       sd f8, -8(r2)
       bne r1, r3, loop
```

prolog

iterate

loop:

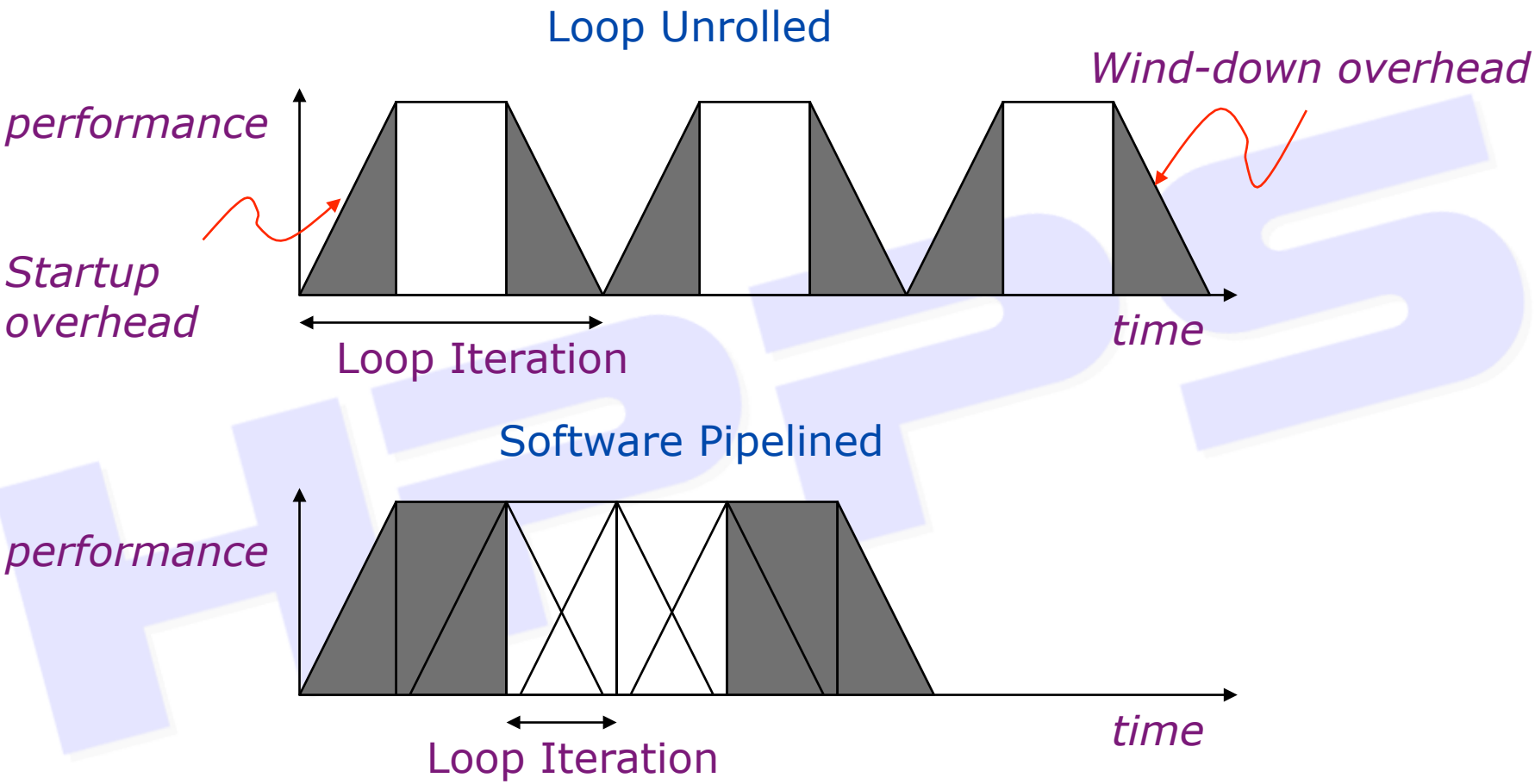
epilog

| | Int1 | Int 2 | M1 | M2 | FP+ | FPx |
|---|---|---|---|---|---|---|
| | | | ld f1 | | | |
| | | | ld f2 | | | |
| | | | ld f3 | | | |
| | add r1 | | ld f4 | | | |
| | | | ld f1 | | fadd f5 | |
| | | | ld f2 | | fadd f6 | |
| | | | ld f3 | | fadd f7 | |
| | add r1 | | ld f4 | | fadd f8 | |
| loop: | | | ld f1 | sd f5 | fadd f5 | |
| | | | ld f2 | sd f6 | fadd f6 | |
| | | add r2 | ld f3 | sd f7 | fadd f7 | |
| | add r1 | bne | ld f4 | sd f8 | fadd f8 | |
| | | | | sd f5 | fadd f5 | |
| | | | | sd f6 | fadd f6 | |
| | | add r2 | | sd f7 | fadd f7 | |
| | | bne | | sd f8 | fadd f8 | |
| | | | | sd f5 | | |

How many FLOPS/cycle?

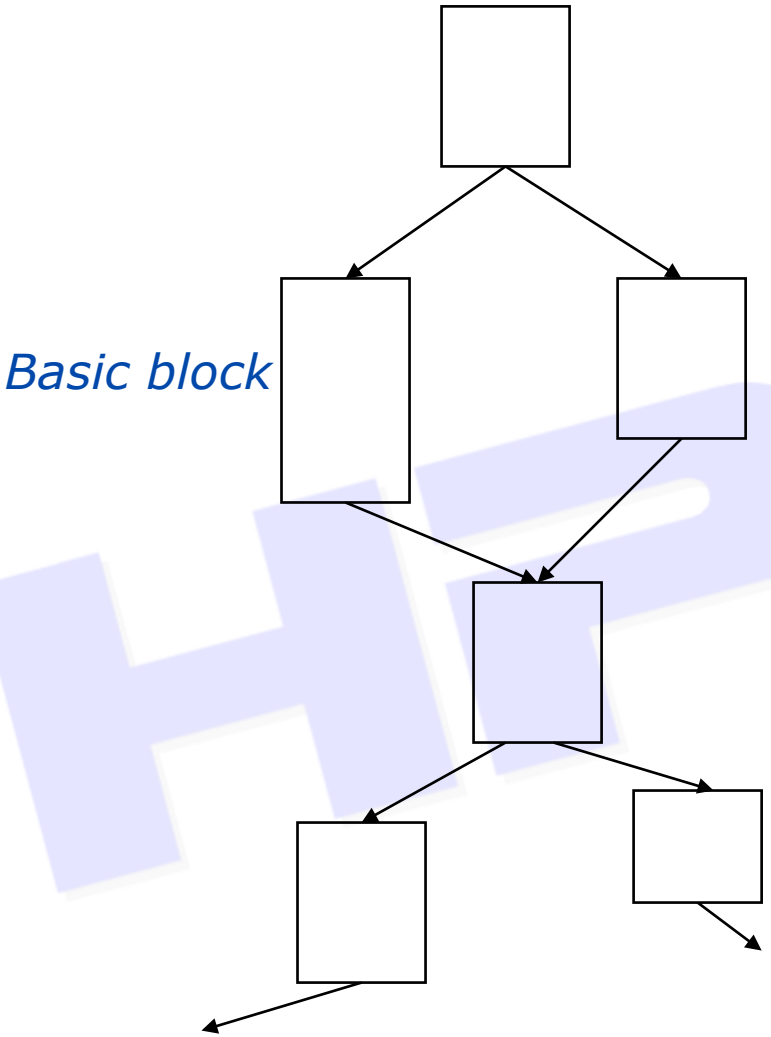
4 fadds / 4 cycles = 1

# Software Pipelining vs. Loop Unrolling




*Software pipelining pays startup/wind-down costs only once per loop, not once per iteration*

# What if there are no loops?

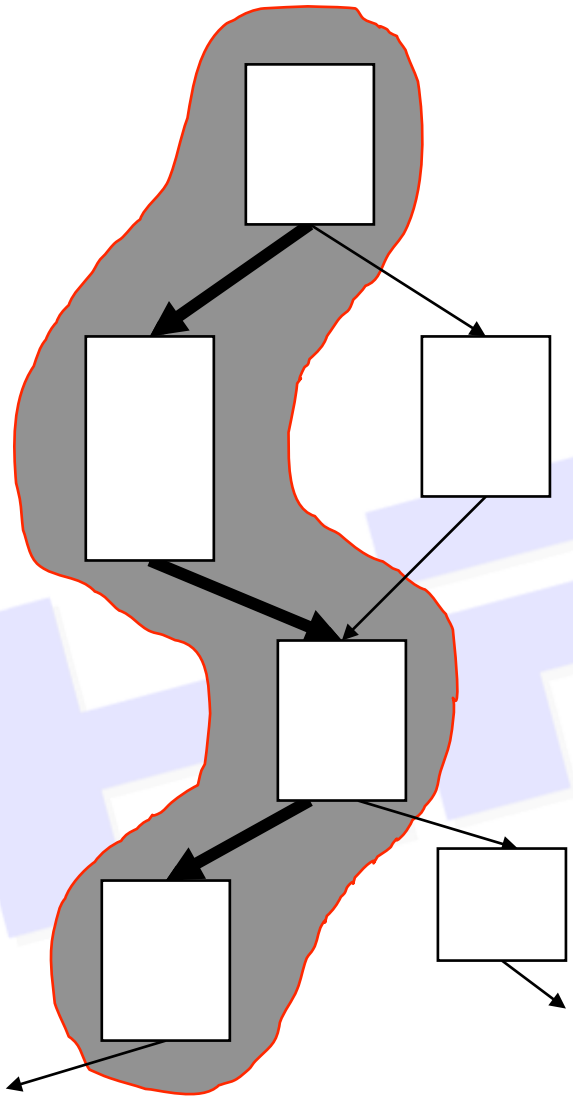



- Branches limit basic block size in control-flow intensive irregular code
- Difficult to find ILP in individual basic blocks

# Trace scheduling

- Trace scheduling focuses on traces
  - A trace is a loop-free sequence of basic blocks embedded in the control flow graph (Fisher)
  - It is an execution path which can be taken for some set of inputs
  - The chances that a trace is actually executed depends on the input set that allows its execution
- Some traces are executed much more frequently than others

# Trace Scheduling *[ Fisher,Ellis]*



- Pick string of basic blocks, a *trace*, that represents most frequent branch path
- Use profiling feedback or compiler heuristics to find common branch paths
- Schedule whole "trace" at once
- Add fixup code to cope with branches jumping out of trace

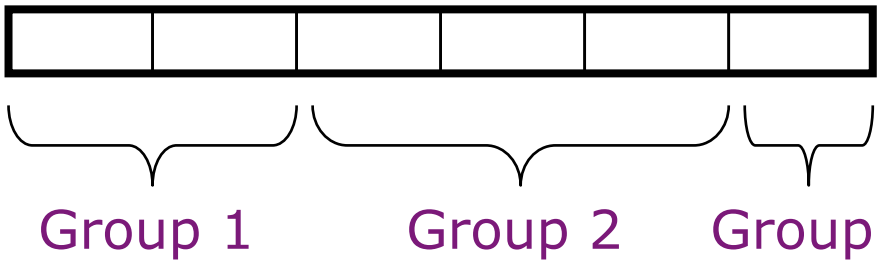
# Trace scheduling and loops

- Trace scheduling and loops
  - Trace scheduling cannot proceed beyond a loop barrier
  - Techniques used to overcome this limitation are based n loop unrolling
- Negative effects on unrolling
  - Unrolling produces much extra code
  - It also looses performance, because of the costs of starting and closing the iterations
- Traces scheduling schedules traces in order of decreasing probability of being executed
  - So, most frequently executed traces get better schedules
  - Traces are scheduled as if they were basic blocks (no special considerations for branches)

# Problems with "Classic" VLIW

- Object-code compatibility
  - have to recompile all code for every machine, even for two machines in same generation
- Object code size
  - instruction padding wastes instruction memory/cache
  - loop unrolling/software pipelining replicates code
- Scheduling variable latency memory operations
  - caches and/or memory bank conflicts impose statically unpredictable variability
- Knowing branch probabilities
  - Profiling requires an significant extra step in build process
- Scheduling for statically unpredictable branches
  - optimal schedule varies with branch path

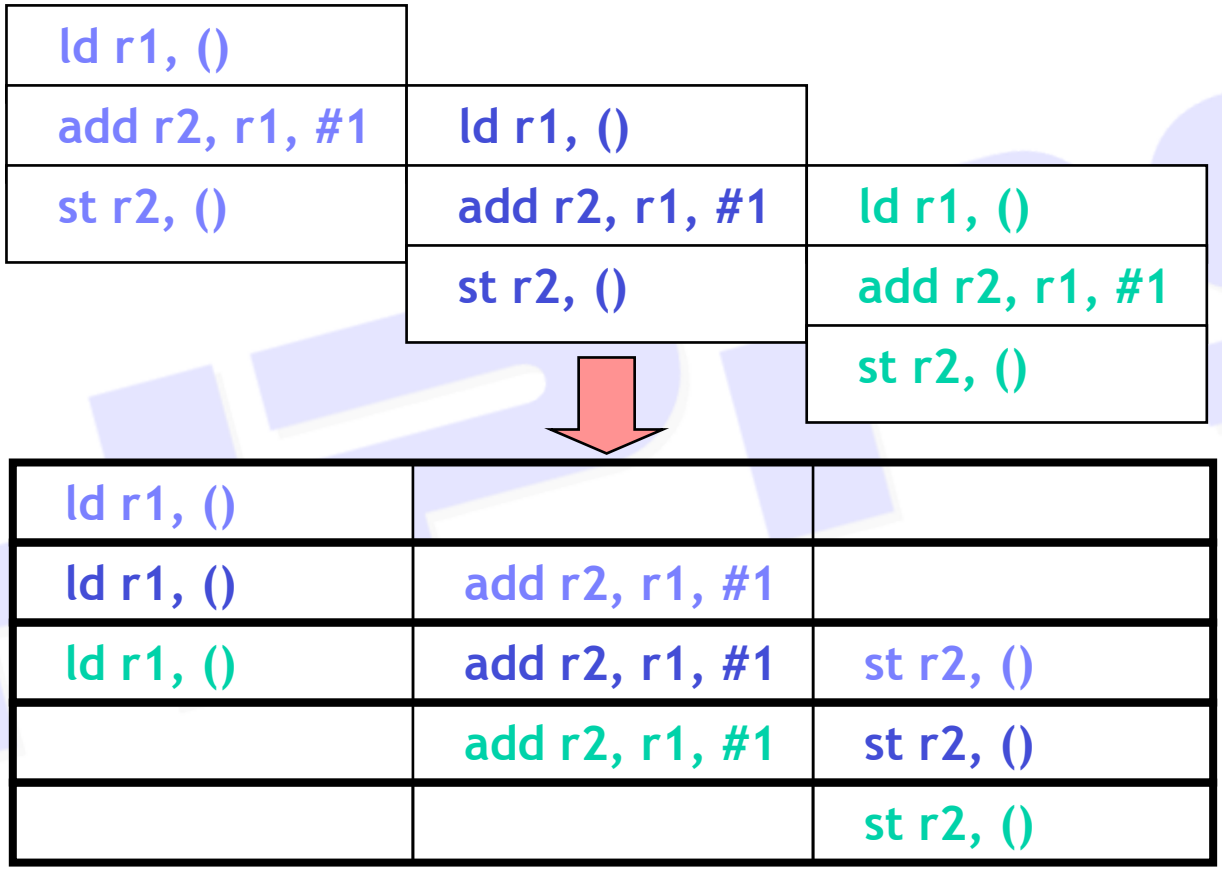
# VLIW Instruction Encoding




- Schemes to reduce effect of unused fields
  - Compressed format in memory, expand on I-cache refill
    - used in Multiflow Trace
    - introduces instruction addressing challenge
  - Mark parallel groups
    - used in TMS320C6x DSPs, Intel IA-64
  - Provide a single-op VLIW instruction
    - Cydra-5 UniOp instructions

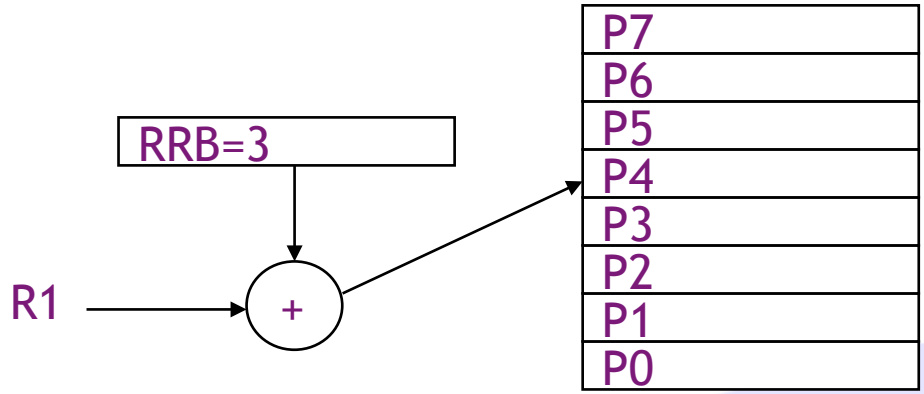
# Rotating Register Files

Problems: Scheduled loops require lots of registers,
Lots of duplicated code in prolog, epilog




Solution: Allocate new set of registers for each loop iteration

# Rotating Register File




Rotating Register Base (RRB) register points to base of current register set. Value added on to logical register specifier to give physical register number. Usually, split into rotating and non-rotating registers.

| | | | |
|---|---|---|---|
| ld r1, () | | | dec RRB |
| ld r1, () | add r3, r2, #1 | | dec RRB |
| ld r1, () | add r3, r2, #1 | st r4, () | bloop |
| | add r2, r1, #1 | st r4, () | dec RRB |
| | | st r4, () | dec RRB |

Prolog

Loop

Epilog

Loop closing branch decrements RRB

# Cydra-5: Memory Latency Register (MLR)

Problem: Loads have variable latency

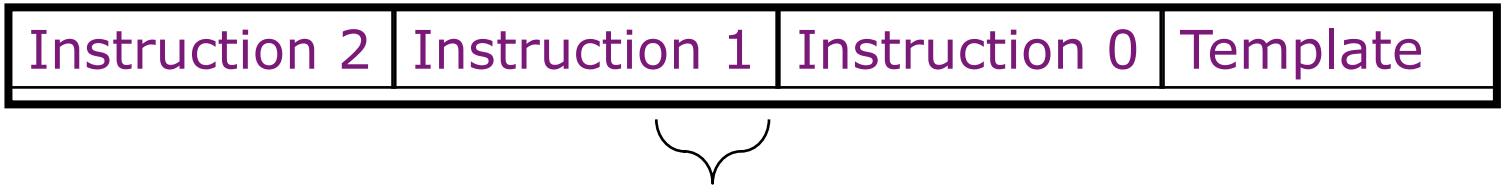
Solution: Let software choose desired memory latency

- Compiler schedules code for maximum load-use distance
- Software sets MLR to latency that matches code schedule
- Hardware ensures that loads take exactly MLR cycles to return values into processor pipeline
  - Hardware buffers loads that return early
  - Hardware stalls processor if loads return late

# Intel EPIC IA-64

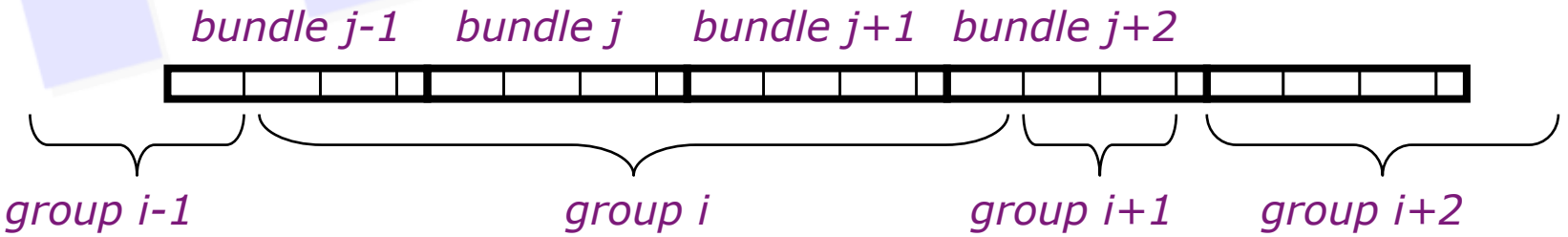
- EPIC is the style of architecture (cf. CISC, RISC)
  - Explicitly Parallel Instruction Computing
- IA-64 is Intel's chosen ISA (cf. x86, MIPS)
  - IA-64 = Intel Architecture 64-bit
  - An object-code compatible VLIW
- Itanium (aka Merced) is first implementation (cf. 8086)
  - First customer shipment expected 1997 (actually 2001)
  - McKinley, second implementation shipped in 2002

# IA-64 Instruction Format




128-bit instruction bundle

- Template bits describe grouping of these instructions with others in adjacent bundles
- Each group contains instructions that can execute in parallel



# IA-64 Registers

- 128 General Purpose 64-bit Integer Registers
- 128 General Purpose 64/80-bit Floating Point Registers
- 64 1-bit Predicate Registers

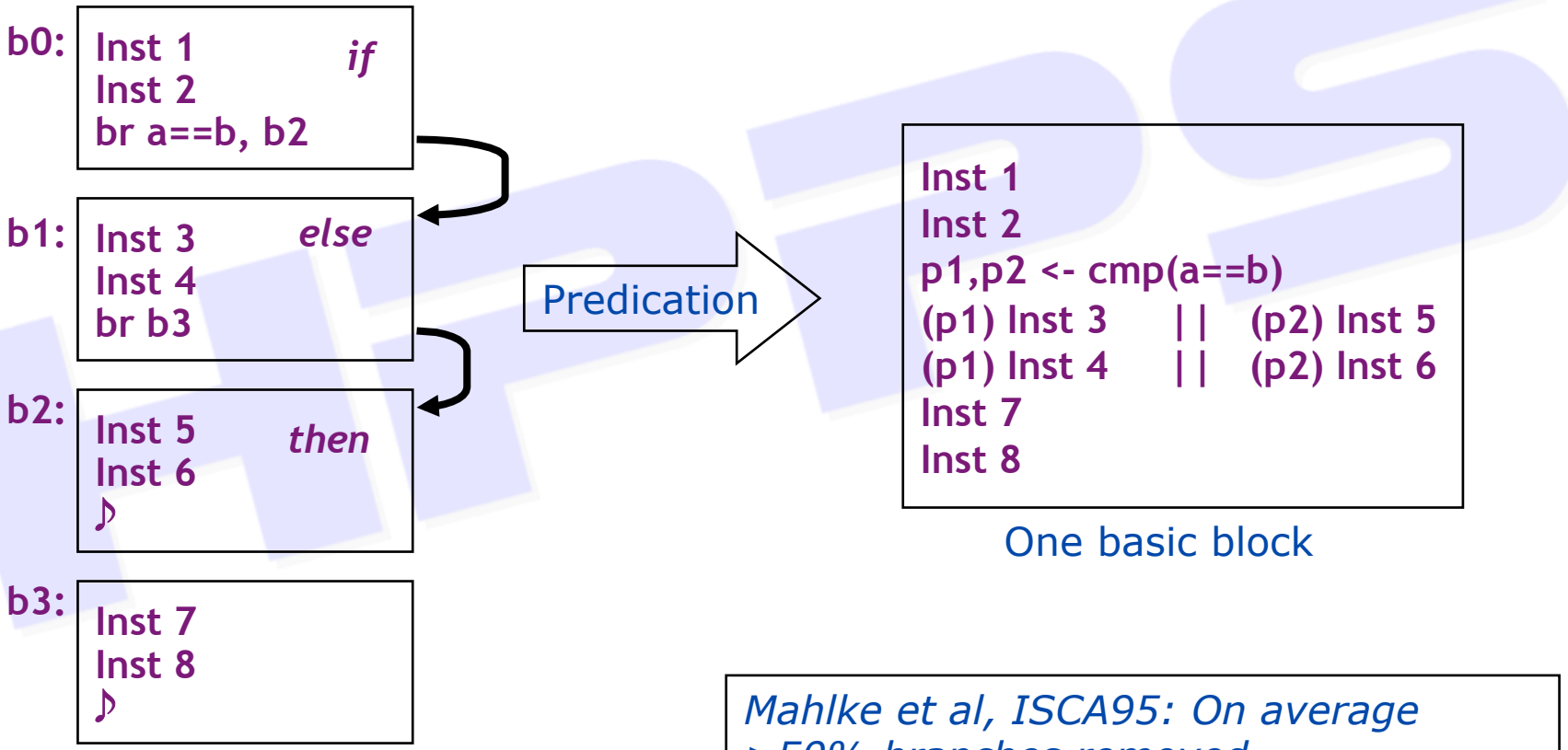
- GPRs rotate to reduce code size for software pipelined loops

# IA-64 Predicated Execution

Problem: Mispredicted branches limit ILP

Solution: Eliminate hard to predict branches with predicated execution

- Almost all IA-64 instructions can be executed conditionally under predicate
- Instruction becomes NOP if predicate register false

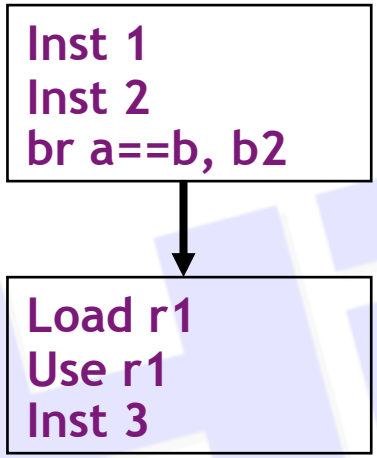



*Mahlke et al, ISCA95: On average >50% branches removed*

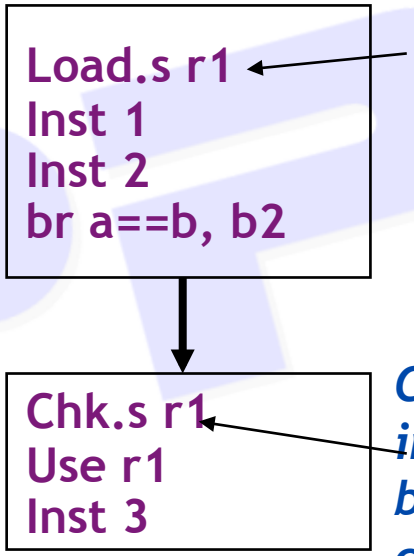
# IA-64 Speculative Execution

Problem: Branches restrict compiler code motion

Solution: Speculative operations that don't cause exceptions




*Can't move load above branch because might cause spurious exception*




*Speculative load never causes exception, but sets "poison" bit on destination register*

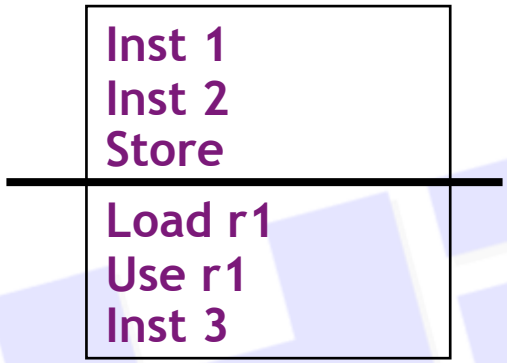
*Check for exception in original home block jumps to fixup code if exception detected*

Particularly useful for scheduling long latency loads early

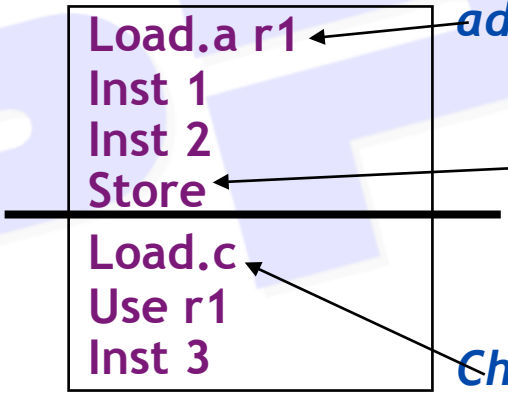
# IA-64 Data Speculation

Problem: Possible memory hazards limit code scheduling

Solution: Hardware to check pointer hazards




*Can't move load above store because store might be to same address*



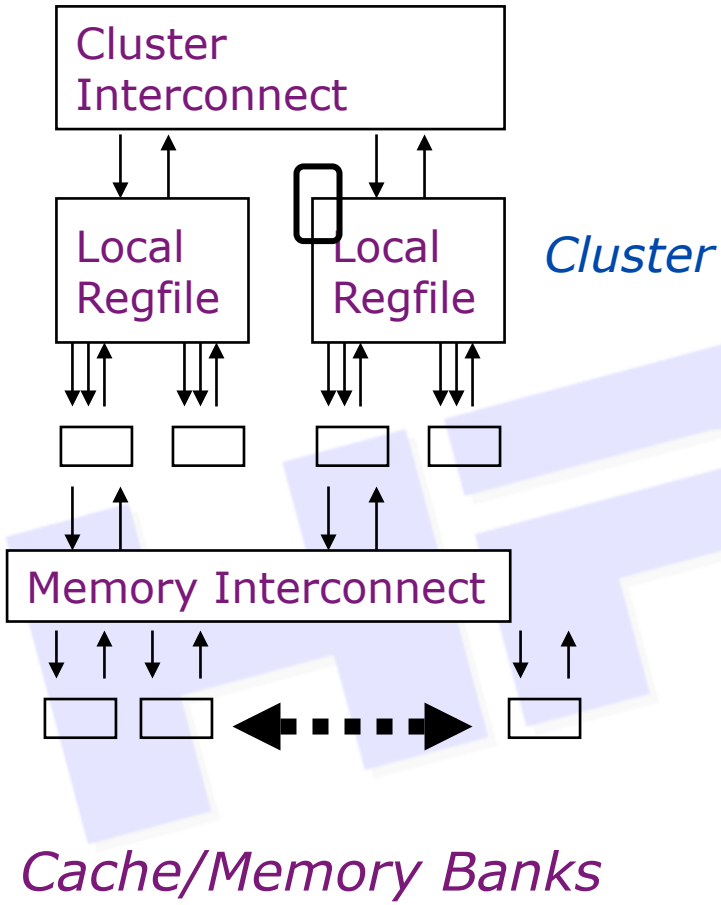

*Data speculative load adds address to address check table*

*Store invalidates any matching loads in address check table*

*Check if load invalid (or missing), jump to fixup code if so*

Requires associative hardware in address check table

# Clustered VLIW




- Divide machine into clusters of local register files and local functional units
- Lower bandwidth/higher latency interconnect between clusters
- Software responsible for mapping computations to minimize communication overhead
- Common in commercial embedded VLIW processors, e.g., TI C6x DSPs, HP Lx processor
- (Same idea used in some superscalar processors, e.g., Alpha 21264)