Static Multiple-Issue Processors: VLIW Approach

Instructor: Prof. Cristina Silvano, email: cristina.silvano@polimi.it

Teaching Assistant: Dr. Giovanni Agosta, email: agosta@acm.org

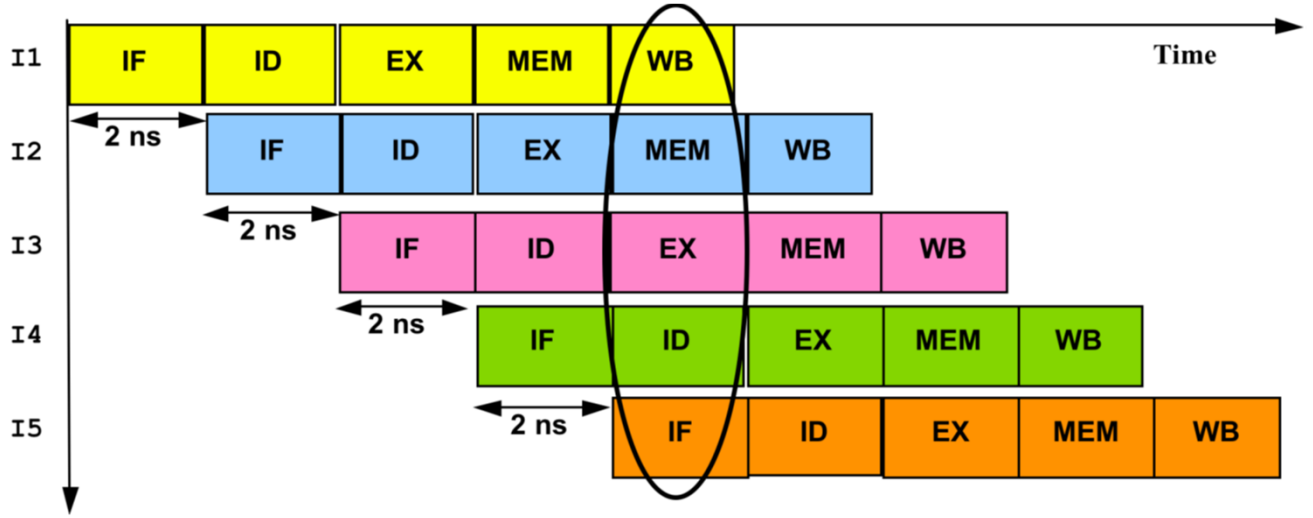
Dipartimento di Elettronica, Informazione e Bioengegneria
Politecnico di Milano

Summary

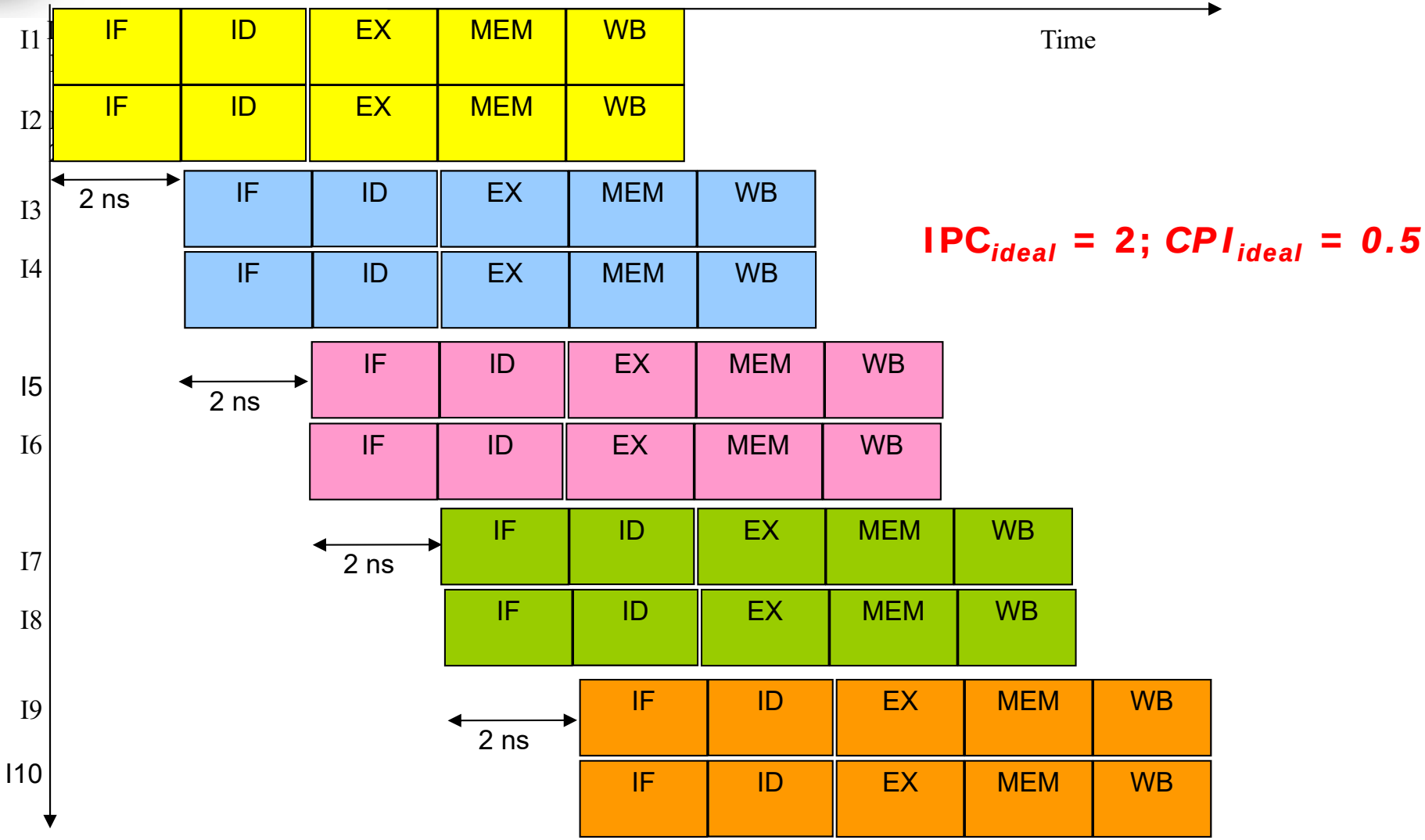
- Recap on Instruction Level Parallelism
- VLIW Architectures
- Code Scheduling for VLIW Architectures
  - *Next lecture*

Recap on Single-Issue Processors

- Until now:
  - RISC architecture
  - MIPS Instruction Pipeline
  - Pipeline Hazards (Instruction Control/Data Dependencies)
  - Branch prediction mechanisms (Dynamic - Static)
- Techniques/Optimizations to exploit ILP on Single Issue Architectures
  - **Ideal CPI = 1** → 1 instruction takes 1 cycle to execute, in best case



Recap on Multiple Issue Processors



$IPC_{ideal} = 2; CPI_{ideal} = 0.5$

Towards Multiple-Issue Processors

- **Single-Issue Processors:** Scalar processors that fetch and issue max one operation in each clock cycle.
- **Multiple-Issue Processors** require:
  - Fetching more instructions in a cycle (higher bandwidth from the instruction cache)
- **Multiple-Issue Processors** can be:
  - **Dynamically scheduled** (issue a varying number of instructions at each clock cycle).
  - **Statically scheduled** (issue a fixed number of instructions at each clock cycle).

Recap on Dynamically Scheduled Processors

- Number of issued instructions per cycle varies **(1-8)**
  - 2+ → Superscalar Processors; CPI = 1/issued instructions
- Scheduling can be performed purely in **hardware**
  - See Tomasulo's algorithm
  - However, the compiler can improve the quality of the schedule
- Superscalar Processors are common in general purpose (desktop/server) computing
  - Most Intel processors from Pentium on
  - Alpha, modern PowePC, Sparc, etc.
- However, hardware techniques such as Tomasulo are costly:
  - PowerPC 750 Instruction Sequencer ~=70% ALU+FPU+LSU
  - Cycle time limited by delay of scheduling logic
  - Design verification becomes expensive

Recap on Dynamic Scheduling

- The **hardware** rearranges 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.

Statically Scheduled Processors

- **Compilers** can use sophisticated algorithms for code scheduling to exploit ILP (Instruction Level Parallelism)
  - Detect whether two instructions can be parallelized
  - Schedule them so they will be executed in parallel
- **Problem:** the amount of parallelism available within a basic block is small (<3 in generic code)
  - **Basic Block:** a straight-line code sequence with no branches in except to the entry and no branches out except at the exit
  - **Example:** For typical MIPS programs the average branch frequency is between 15% and 25% $\Rightarrow$ from 4 to 7 instructions execute between a pair of branches

Statically Scheduled Processors (2)

- **Data dependences** can further limit the amount of ILP we can exploit within a basic block to much less than the average basic block size
  - True data dependences force sequential execution of instructions
  - The compiler can, however, deal to some extent with false data dependences
- To obtain substantial performance enhancements, we must **exploit ILP across multiple basic blocks** (i.e. across branches).

Dependences

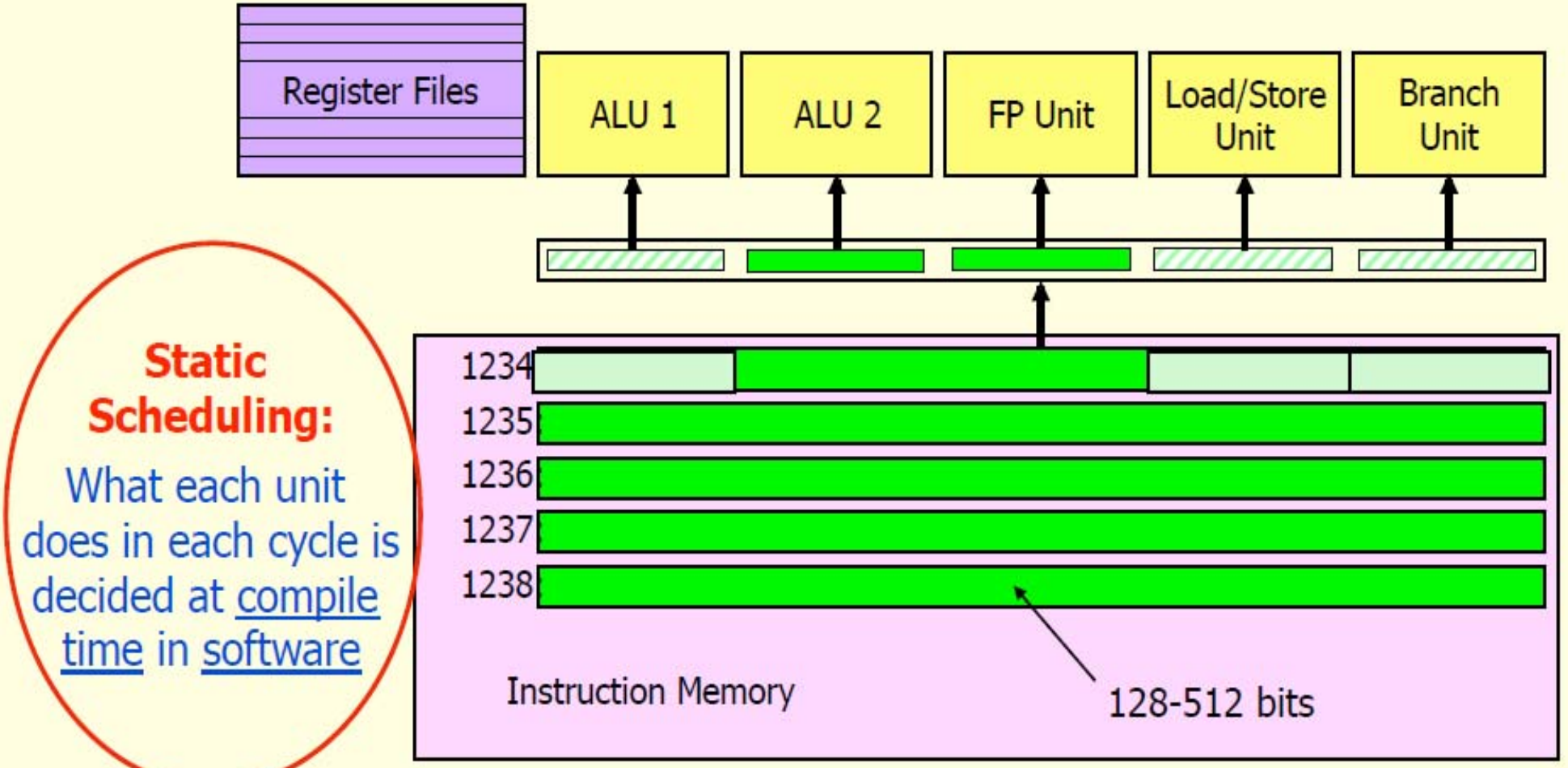
- Determining **dependences** among instructions is critical to defining the amount of parallelism existing in a program.
- If two instructions are dependent, they cannot execute in parallel: they must be executed in order or only partially overlapped.
- Three different types of dependences:
  - **Data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

Program Properties

- **Two properties** are critical to program correctness (and normally preserved by maintaining both data and control dependences):
  - **Data flow:** Actual flow of data values among instructions that produces the correct results and consumes them.
  - **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.

VLIW Processors:
An Alternative Way of Extracting ILP

(Statically Scheduled) Very Long Instruction Word Processor (VLIW)



Static Scheduling:

What each unit does in each cycle is decided at compile time in software

Very Long Instruction Word Processors

- The **single issue packet (*bundle*)** represents a wide instruction (64, 128 or more bits) with multiple independent operations (or syllables) per instruction
  - Thus the name *"Very Long Instruction Word"*
- The compiler identifies statically the multiple independent operations to be executed in parallel by the multiple Functional Units.
- The compiler statically solves the **structural hazards** for the use of the HW resources and the **data hazards.**

VLIW Processors

- VLIW approach advantages:
  - Simpler hardware because the complexity of the control unit is moved to the compiler
  - Low power consumption
  - Good performance through extensive compiler optimization
- VLIW approach disadvantages:
  - Early VLIWs were quite rigid in the instruction format and they required recompilation of programs for different versions of the hardware

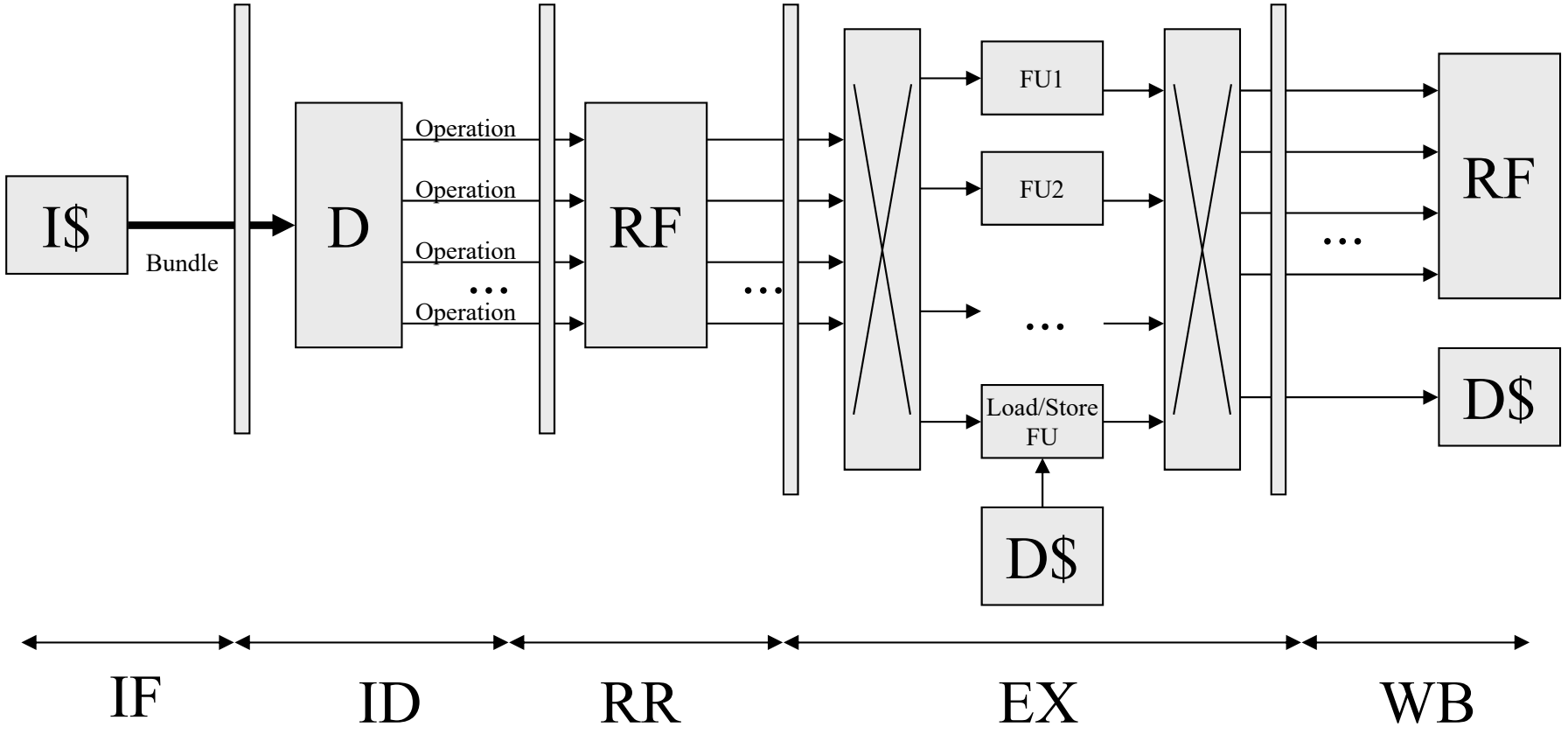
VLIW processors

- The long instruction *(bundle***)** has a set of **operations** *(slots)* for each Functional Unit: for example 2, 4, 5 slots.
- Example: a **5-issue VLIW** has a long instruction that can contain up to *5 operations* (corresponding to *5 slots*) including 2 integer operations, 1 floating point op, 1 memory reference and 1 branch.
- Decode unit is reduced to simple decode logic (for each op): each op is passed to the corresponding FU to be executed
- If there are more parallel FUs than issues, it is eventually present a dispatch network that redirects ops and related operands to FUs.

VLIW processors

- To keep busy the FUs, there must be enough parallelism in the source code to fill in the available operation slots. Otherwise NOPs are inserted.
- There is a shared Register File: If the bundle has *n slots* => there are *2n* read ports and *n* write ports.
- For each bundle, there might be only **one branch** to modify the control flow.
- To keep in-order execution, the Write Back phase of the parallel ops in a bundle must occur at the same clock cycle, to avoid structural hazards accessing the RF and WAR/WAW hazards
- Ops in a bundle are constrained to the latency of the longer latency op in the bundle

Pipelined VLIW Architecture Overview



VLIW Processors: Operation Latency

- VLIW instructions are not atomic, i.e., **latency of operations is exposed to the compiler:**

```
I     [C=A*B, ....];
I+1   [NOP,   ....]; compiler inserted
I+2   [X=C*F, ....];
```

- Assuming multiplication has a latency=2
  - The compiler must schedule the use of C after 1 instruction
  - Otherwise correct execution is compromised
  - True even in the case of a pipelined multiplier

VLIW Processors: Dependences

- True, anti and output dependencies are solved by the compiler and not by the hardware by taking into account FU latency
- **RAW** Hazards:
  - Scalars/superscalars generate NOPs/stalls or execute successive instructions (dynamic scheduling or instruction reordering)
  - In VLIW other instructions are statically inserted by the compiler during the scheduling phase
    - Ideally, instructions not involved in the dependencies should be used
    - Otherwise, the compiler can generate **NOPs**

VLIW Processors: Dependences

- **WAR** and **WAW** hazards are statically solved by the compiler by correctly selecting temporal slots for the operations or by register renaming.
- **Structural hazards** are also solved by the compiler.
- Compiler can provide useful hints on how to statically predict **branches.**
- **Control hazards** are solved dynamically by the hardware by flushing the execution of instructions due to a mispredicted branch.

VLIW Processors: Dynamic events

- The compiler does not know the behavior of some dynamic events such as:
  - **Data Cache Miss:** Stalls are introduced at runtime (latency of a data cache hit is known at compile time)
  - **Branch Misprediction:** need of flushing the execution of instructions in the pipeline.

VLIW scheduling: a simple example

INSTRUCTIONS	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13
L1: lw$2,BASEA($4)	IF	ID	EX	M	WB								
addi $2,$2,INC1		X	IF	ID	EX	M	WB						
lw $3,BASEB($4)				IF	ID	EX	M	WB					
addi $3,$3,INC2					X	IF	ID	EX	M	WB			
addi $4, $4, 4							IF	ID	EX	M	WB		
bne $4,$7, L1								X	IF	ID	EX	M	WB

MIPS schedule with forwarding

VLIW scheduling: a simple example

INSTRUCTIONS	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13
L1: lw$2,BASEA($4)	IF	ID	EX	M	WB								
addi $2,$2,INC1		X	IF	ID	EX	M	WB						
lw $3,BASEB($4)				IF	ID	EX	M	WB					
addi $3,$3,INC2					X	IF	ID	EX	M	WB			
addi $4, $4, 4							IF	ID	EX	M	WB		
bne $4,$7, L1								X	IF	ID	EX	M	WB

4-issue VLIW:
LD/ST latency 2,
INT latency 1,
INT to BR latency 2

SLOT1: LD/ST Ops	SLOT2: LD/ST Ops	SLOT3: Integer Ops	SLOT4: Integer+Branch Ops
lw$2,BASEA($4)	lw $3,BASEB($4)	NOP	NOP
NOP	NOP	NOP	NOP
NOP	NOP	addi $2,$2,INC1	addi $3,$3,INC2
NOP	NOP	addi $4, $4, 4	NOP
NOP	NOP	NOP	NOP
NOP	NOP	NOP	bne $4,$7, L1

Advantages of VLIW

- Compiler can use sophisticated algorithms to schedule code (exploiting *program parallelism*) to increase performance
  - The compiler can observe the code on a much wider instruction window than the hardware
  - The compiler has more time to analyze dependencies and to extract more parallelism
- Instructions have fixed fields $\Rightarrow$ easier to decode
- Reduced hardware complexity
  - Small die area
  - Low Processor Cost
  - Low Power Consumption
  - Easily extended to larger number of FUs

Open Challenges of VLIW Technology

- Need for strong Compiler Technology
  - Detect and exploit parallelism
  - Need to manage parallelism beyond the basic block
- Code Size Increase
  - Explicit NOPs may be numerous
  - Possible solution: *code compression* techniques
    - Used in modern VLIWs, but require added circuitry
- Huge number of registers needed for register renaming
  - Complexity of register file to FU is increased
  - Possible solution: *clustered* VLIWs

Open Challenges of VLIW Technology

- **Binary Incompatibility**
  - Architectures with same ISA but different VLIW bundle size are incompatible
  - Also, same ISA and same VLIW bundle size, but different number and types of FUs and FU latencies → still incompatible!
  - There is no fully satisfactory solution to this issue
  - *Just In Time Compilation* is a possible, if costly, solution → attempted in Transmeta Crusoe
  - In most cases, VLIW are simply employed in embedded systems, where binary compatibility is less important

Early VLIW Architecture

- Multiflow Trace (1987)
  - Designed by Josh Fisher
  - Based on Fisher's ideas for scheduling algorithms
    - Trace scheduling
  - Has several direct descendants
- Cydrome Cydra 5 (1987)
  - The numeric processor for this departmental supercomputer was a VLIW

Modern VLIW Architectures

- Intel Itanium IA-64 EPIC → only major general purpose VLIW processor
- Transmeta Crusoe → designed for low-power general purpose processing (laptops), using binary to binary compilation for x86 compatibility
- STMicroelectronics ST200 (based on HP/STM Lx) → digital video processing
  - STxP70 can also be configured as a 2-way VLIW
- AMD Radeon R600 series → unified shaders using VLIW cores
  - Not used in later series aiming at more GPGPU loads

Conclusions

- VLIW Architectures employ statically scheduled multiple issues to:
  - Reduce hardware complexity
  - Decrease the CPI by exploiting ILP
- VLIW Architectures are more suitable for application specific processors for use in embedded systems
  - Digital video processing
  - Graphics
- VLIW architectures face significant limitations when dealing with legacy code, since poor binary portability
- VLIW architectures strongly depend on compiler quality