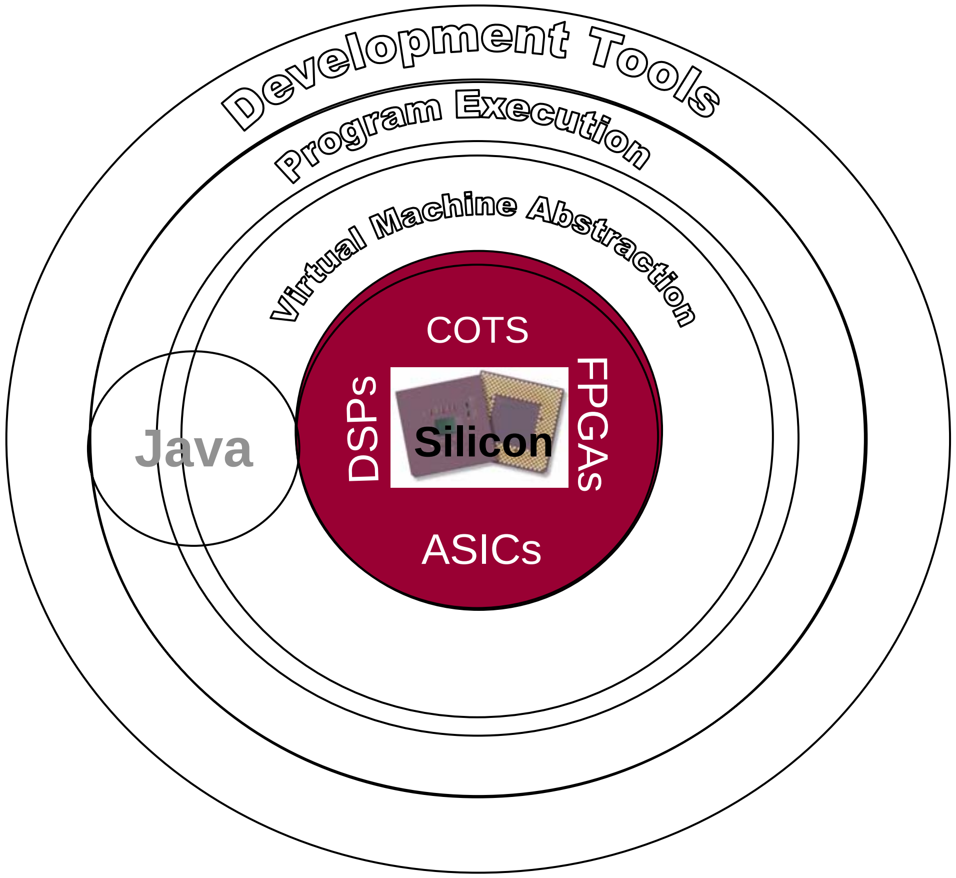
Topic 1
Evolution of ILP in Microprocessors

The slides used for this lecture were contributed

by Prof. B. Narahari, George Washington University

Where Superscalar vs VLIW Takes Place



- The innermost circle represents the various types of architectures: VLIW and Superscalar

Introduction to ILP

- What is ILP?
  - Processor and Compiler design techniques that speed up execution by causing individual machine operations to execute in parallel
- ILP is transparent to the user
  - Multiple operations executed in parallel even though the system is handed a single program written with a sequential processor in mind
- Same execution hardware as a normal RISC machine
  - May be more than one of any given type of hardware

Why ILP for Embedded Processors?

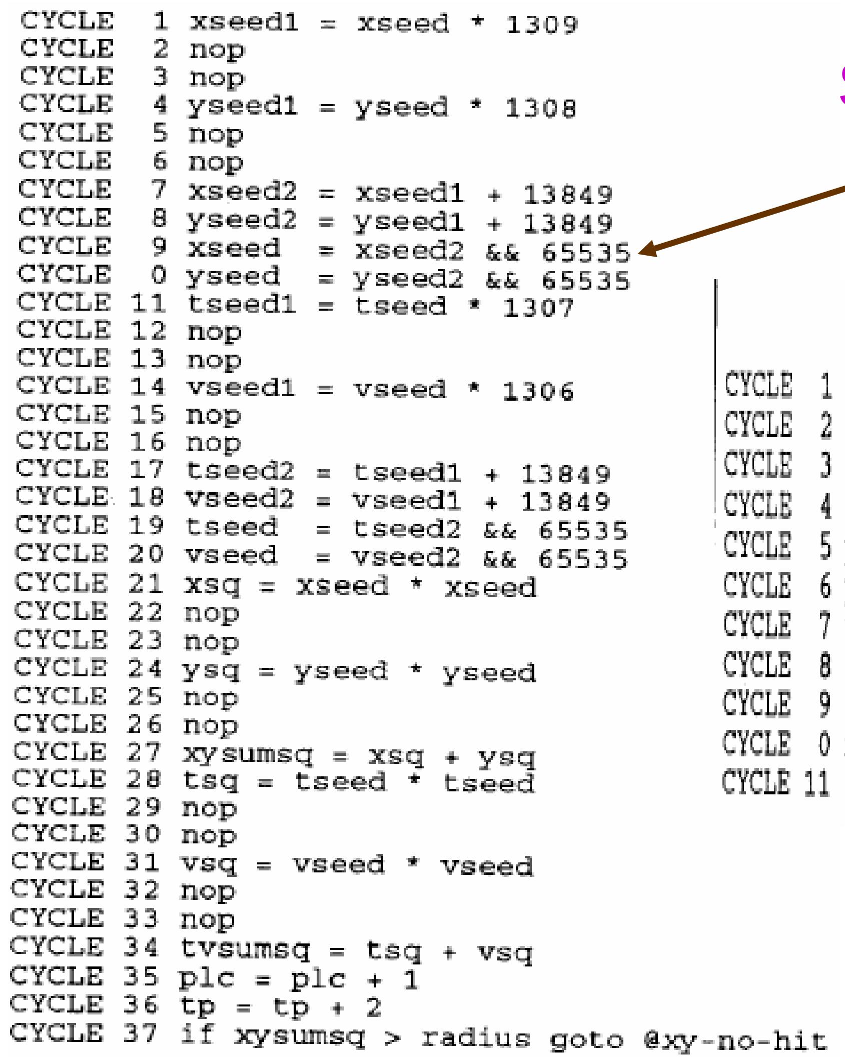
- Current state-of-art leverages RISC pipeline technology e.g. ARM
- Next logical progression for increased performance is some level of parallelism
  - Constraints of embedded systems prohibit multiprocessor solutions I.e. power and size constraints
  - Instruction level parallelism is feasible and offers improved performance
- Some current embedded applications use ILP processor technology in application specific domains I.e. DSP

Example Execution

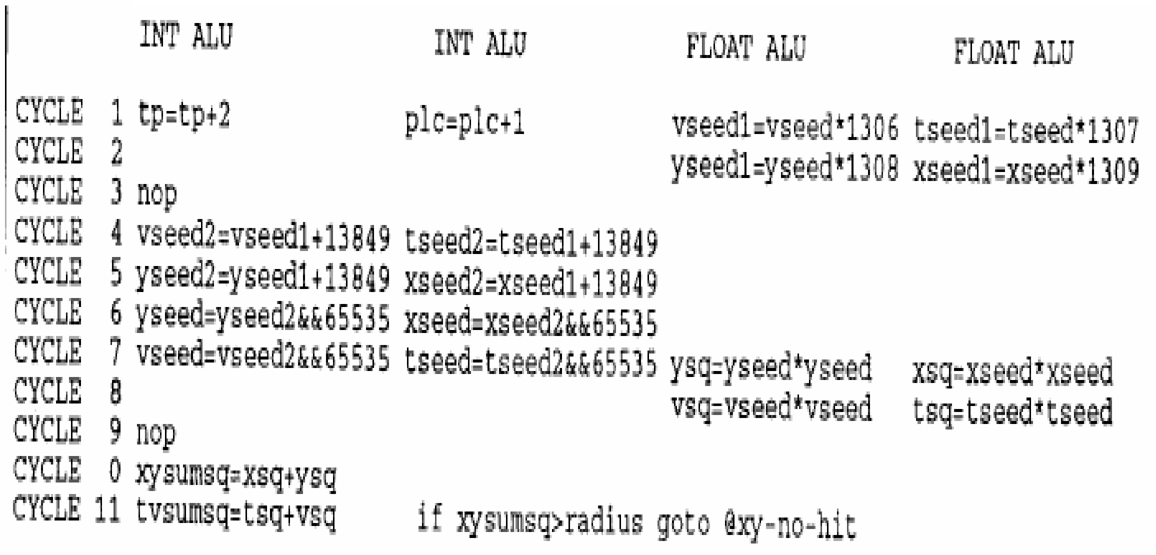
Functional Unit	Operations Performed	Latency
Integer Unit 1	Integer ALU Operations Integer Multiplication Loads Stores	1 2 2 1
Integer Unit 2 / Branch Unit	Integer ALU Operations Integer Multiplication Loads Stores Test-and-branch	1 2 2 1 1
Floating-point Unit 1 Floating-point Unit 2	Floating Point Operations	3

Example Execution

Sequential Execution



```
CYCLE  1 xseed1 = xseed * 1309
CYCLE  2 nop
CYCLE  3 nop
CYCLE  4 yseed1 = yseed * 1308
CYCLE  5 nop
CYCLE  6 nop
CYCLE  7 xseed2 = xseed1 + 13849
CYCLE  8 yseed2 = yseed1 + 13849
CYCLE  9 xseed  = xseed2 && 65535
CYCLE  0 yseed  = yseed2 && 65535
CYCLE 11 tseed1 = tseed * 1307
CYCLE 12 nop
CYCLE 13 nop
CYCLE 14 vseed1 = vseed * 1306
CYCLE 15 nop
CYCLE 16 nop
CYCLE 17 tseed2 = tseed1 + 13849
CYCLE 18 vseed2 = vseed1 + 13849
CYCLE 19 tseed  = tseed2 && 65535
CYCLE 20 vseed  = vseed2 && 65535
CYCLE 21 xsq = xseed * xseed
CYCLE 22 nop
CYCLE 23 nop
CYCLE 24 ysq = yseed * yseed
CYCLE 25 nop
CYCLE 26 nop
CYCLE 27 xysumsq = xsq + ysq
CYCLE 28 tsq = tseed * tseed
CYCLE 29 nop
CYCLE 30 nop
CYCLE 31 vsq = vseed * vseed
CYCLE 32 nop
CYCLE 33 nop
CYCLE 34 tvsumsq = tsq + vsq
CYCLE 35 plc = plc + 1
CYCLE 36 tp = tp + 2
CYCLE 37 if xysumsq > radius goto @xy-no-hit
```



	INT ALU	INT ALU	FLOAT ALU	FLOAT ALU
CYCLE 1	tp=tp+2	plc=plc+1	vseed1=vseed*1306	tseed1=tseed*1307
CYCLE 2			yseed1=yseed*1308	xseed1=xseed*1309
CYCLE 3	nop			
CYCLE 4	vseed2=vseed1+13849	tseed2=tseed1+13849		
CYCLE 5	yseed2=yseed1+13849	xseed2=xseed1+13849		
CYCLE 6	yseed=yseed2&&65535	xseed=xseed2&&65535		
CYCLE 7	vseed=vseed2&&65535	tseed=tseed2&&65535	ysq=yseed*yseed	xsq=xseed*xseed
CYCLE 8			vsq=vseed*vseed	tsq=tseed*tseed
CYCLE 9	nop			
CYCLE 0	xysumsq=xsq+ysq			
CYCLE 11	tvsumsq=tsq+vsq	if xysumsq>radius goto @xy-no-hit		

ILP Execution

Early History of ILP

- 1940s and 1950s
  - Parallelism first exploited in the form of horizontal microcode
    - **Wilkes and Stringer, 1953 - "In some cases it may be possible for two or more micro-operations to take place at the same time"**
- 1960s - Transistorized computers
  - More gates available than necessary for a general-purpose CPU
  - ILP provided at machine-language level

Early History of ILP

- 1963 - Control Data Corporation delivered the CDC 6600
  - 10 functional units
  - Any unit could begin execution in a given cycle even if other units were still processing data-independent earlier operations
- 1967 - IBM delivered the 360/91
  - Fewer functional units than the CDC 6600
  - Far more aggressive in its attempts to rearrange the instruction stream to keep functional units busy

References

- “Instruction-Level Parallel Processing: History, Overview and Perspective”, B. Ramakrishna Rau and Joseph A. Fisher, October 1992
- “Instruction-Level Parallel Processing”, Joseph A. Fisher and B. Ramakrishna Rau, January 1992

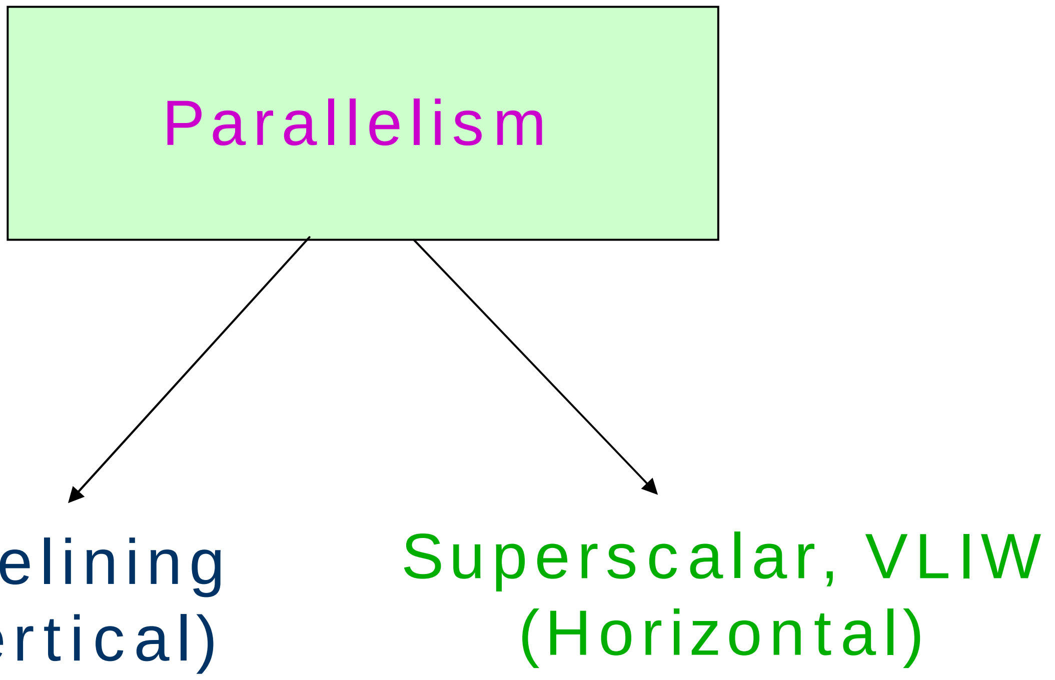
Recent History of ILP

- 1970s - Specialized Signal Processing Computers
  - Horizontally microcoded FFTs and other algorithms
- 1980s - Speed Gap between writeable and read-only memory narrows
  - Advantages of read-only control store began to disappear
  - General purpose microprocessors moved toward RISC concept.
  - Specialized processors provided writeable control memory, giving users access to ILP
    - **called Very Long Instruction Word (VLIW)**

Recent History of ILP

- 1990s - More silicon than necessary for implementation of a RISC microprocessor
  - Virtually all designs take advantage of the available real estate by providing some form of ILP
    - **Primarily in the form of superscalar capability**
    - **Some have used VLIWs as well**

ILP Processors



Instruction Level Parallel(ILP) Processors

- Early ILP - one of two orthogonal concepts:
  - Pipelining(RISC)
  - Multiple (non-pipelined) units
- Progression to multiple pipelined units
- Instruction issue became bottleneck, led to
  - Superscalar ILP processors
  - Very Large Instruction Word (VLIW)
- Embedded systems exploit ILP to improve performance

ILP Processors

- Whereas pipelined processors work like an assembly line
- VLIW and Superscalar processors operate basically in parallel, making use of a number of concurrently working execution units (EU)
- There is a natural progression from pipelined processors to VLIW/Superscalar processors in the embedded systems community.

Questions Facing ILP System Designers

- What gives rise to instruction-level parallelism in conventional, sequential programs and how much of it is there?
- How is the potential parallelism identified and enhanced?
- What must be done in order to exploit the parallelism that has been identified?
- How should the work of identifying, enhancing and exploiting the parallelism be divided between the hardware and the compiler?
- What are the alternatives in selecting the architecture of an ILP processor?

ILP Architectures

- Between the compiler and the run-time hardware, the following functions must be performed
  - Dependencies between operations must be determined
  - Operations that are independent of any operation that has not yet completed must be determined
  - Independent operations must be scheduled to execute at some particular time, on some specific functional unit, and must be assigned a register into which the result may be deposited.

ILP Architecture Classifications

- Sequential Architectures
  - The program is not expected to convey any explicit information regarding parallelism
- Dependence Architectures
  - The program explicitly indicates dependencies between operations
- Independence Architectures
  - The program provides information as to which operations are independent of one another

Sequential Architecture

- Program contains no explicit information regarding dependencies that exist between instructions
- Dependencies between instructions must be determined by the hardware
  - It is only necessary to determine dependencies with sequentially preceding instructions that have been issued but not yet completed
- Compiler may re-order instructions to facilitate the hardware’s task of extracting parallelism

Sequential Architecture Example

- Superscalar processor is a representative ILP implementation of a sequential architecture
  - For every instruction issued by a Superscalar processor, the hardware must check whether the operands interfere with the operands of any other instruction that is either
    - **Already in execution**
    - **Have been issued but are waiting for the completion of interfering instructions that would have been executed earlier in a sequential program**
    - **Is being issued concurrently but would have been executed earlier in the sequential execution of the program**

Sequential Architecture Example

- Superscalar processors attempt to issue multiple instructions per cycle
  - However, essential dependencies are specified by sequential ordering so operations must be processed in sequential order
  - This proves to be a performance bottleneck that is very expensive to overcome
- Alternative to multiple instructions per cycle is pipelining and issue instructions faster

Dependence Architecture

- Compiler or programmer communicates to the hardware the dependencies between instructions
  - Removes the need to scan the program in sequential order (the bottleneck for superscalar processors)
- Hardware determines at run-time when to schedule the instruction

Dependence Architecture Example

- Dataflow processors are representative of Dependence architectures
  - Execute instruction at earliest possible time subject to availability of input operands and functional units
  - Dependencies communicated by providing with each instruction a list of all successor instructions
  - As soon as all input operands of an instruction are available, the hardware fetches the instruction
  - The instruction is executed as soon as a functional unit is available
- Few Dataflow processors currently exist

Independence Architecture

- By knowing which operations are independent, the hardware needs no further checking to determine which instructions can be issued in the same cycle
- The set of independent operations is far greater than the set of dependent operations
  - Only a subset of independent operations are specified
- The compiler may additionally specify on which functional unit and in which cycle an operation is executed
  - The hardware needs to make no run-time decisions

Independence Architecture Example

- VLIW processors are examples of Independence architectures
  - Specify exactly which functional unit each operation is executed on and when each operation is issued
  - Operations are independent of other operations issued at the same time as well as those that are in execution
  - Compiler emulates at compile time what a dataflow processor does at run-time

Independence Architecture Example

- Horizon
  - Encodes an integer H with each operation and guarantees that the next H operations are data-independent of the current operation
  - The hardware simply insures that no more than H subsequent operations will be released before the current operation completes

ILP Architecture Comparison

	Sequential Architecture	Dependence Architecture	Independence Architecture
Additional information required in the program	None	Complete specification of dependencies between operations	Minimally, a partial list of independencies. Typically, a complete specification of when and where each operation is to be executed
Typical ILP Processor	Superscalar	Dataflow	VLIW
Analysis of dependencies between operations	Performed by hardware	Performed by compiler	Performed by compiler
Analysis of independent operations	Performed by hardware	Performed by hardware	Performed by compiler
Final operation scheduling	Performed by hardware	Performed by hardware	Typically, performed by compiler
Role of compiler	Rearranges code to make the analysis and scheduling hardware more successful	Replaces some analysis hardware	Replaces virtually all the analysis and scheduling hardware

What does this mean for Embedded Systems?

- ASICs and DSPs have been typically designed with RISC and VLIW characteristics.
- Embedded systems are moving away from pipelined RISC architectures to improve performance.
- Microprocessor technology is offering superscalar and VLIW as solutions for embedded systems.

VLIW and Superscalar

- Basic structure of VLIW and superscalar consists of a number of EUs, capable of parallel operation on data fetched from a register file
- VLIW and superscalar processors require highly multiported register files
  - limit on register ports places inherent limitation on maximum number of EUs

Contrasting VLIW & Superscalar

- Presentation of instructions:
  - VLIW receive multi-operation instructions
  - Superscalar receive traditional sequential stream
- VLIW needs very long instructions in order to specify what each EU should do
- Superscalar parallelize a sequential stream of conventional instructions

Contrasting VLIW & Superscalar

- VLIW processors expect dependency free code on each cycle whereas superscalars do not
  - Superscalars cope with dependencies using hardware (dynamic instruction scheduling)
  - VLIW lets the compiler cope with dependencies (static instruction scheduling)
- Decode and Issue unit in superscalar processors issue multiple instructions for the EUs per cycle

Superscalar Processors

- Runtime or dynamic tasks:
  - **parallel decoding**
  - **superscalar instruction issue**
  - **parallel instruction execution**
    - **preserving sequential consistency of exception processing**

Superscalar: Parallel Decoding

- Scalar processor decodes one instruction/cycle
- Superscalar decodes multiple instructions per cycle
- Check for *dependencies*
  - With respect to instructions currently executing
  - With respect to candidate instructions for issue
  - Since more instructions are in execution, more comparisons to be performed
- Requires complex HW to support the dynamic scheduling

Superscalar: Parallel Execution

- When instructions are executed in parallel they might finish out of program order
  - unequal execution times
- Specific means needed to preserve logical consistency
  - preservation of sequential consistency
- Exceptions during execution
  - preserve sequential consistency of exception processing
- Finishing out of order can be avoided with multiple EU -- how ?
  - delay result delivery to visible registers
- Superscalar hardware is power-inefficient compared to VLIW!
  - Of great concern to embedded systems design

VLIW Processors

- Length (number of bits) of VLIW instruction depends on two factors:
  - Number of EUs and
  - Lengths required for controlling each of the EUs
- Static scheduling removes burden of instruction scheduling from processor
  - Reduces complexity of processor at a greater than linear rate
  - Lesser complexity can be exploited either by increasing the clock rate or degree of parallelism
  - Helps sustain Moore’s Law

VLIW Tradeoffs

- Compiler takes full responsibility for dependency resolution and parallelism
- This implies architecture has to be exposed in some detail to compiler
  - Number and types of EU, their latencies, memory load-use delays etc.
  - Compiler has to be aware of technology dependent parameters like latencies!

VLIW Tradeoffs - Cont'd

- Mispredicted memory latencies lead to cache misses
  - Compiler must take into account worst case delay values
  - This leads to performance degradation
- VLIW uses long instruction words
  - Some of the fields in the instruction word may not be used
  - No-ops
  - Wasted memory space and memory bandwidth

Summary

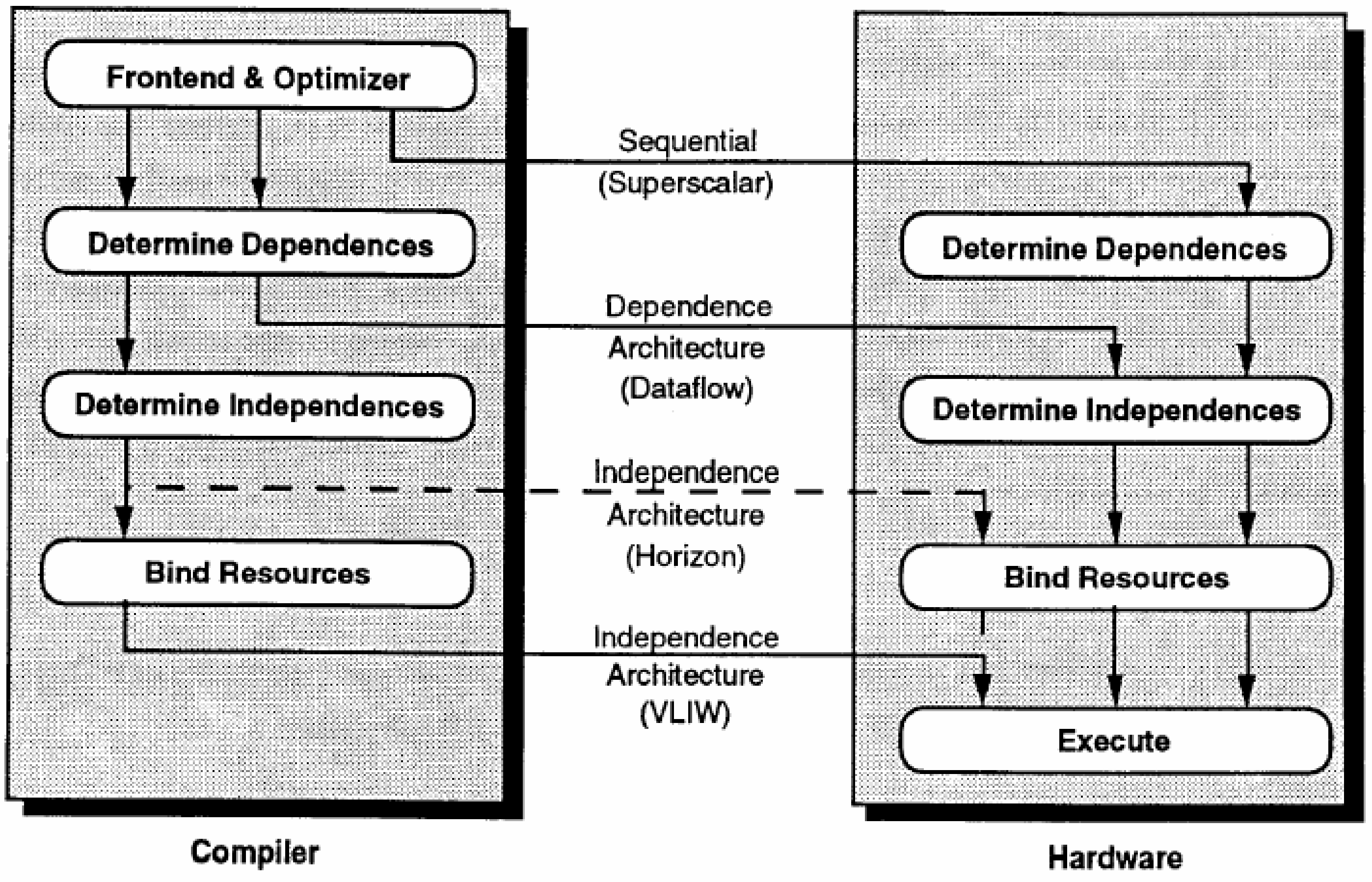


Figure 2. Division of responsibilities between the compiler and the hardware for the three classes of architecture.

Embedded Processors

Instruction Scheduling

- dependencies must be detected and resolved
- instructions that are not dependent on each other must be scheduled
- *static:* accomplished by compiler which avoids dependencies by rearranging code
- *dynamic:* detection and resolution performed by hardware. processor typically maintains issue window (prefetched inst) and execution window (being executed). check for dependencies in issue window.

More Hardware Features to Support ILP

- Pipelining
  - Advantages
    - Relatively low cost of implementation - requires latches within functional units
    - With pipelining, ILP can be doubled, tripled or more
  - Disadvantages
    - Adds delays to execution of individual operations
    - Increased latency eventually counterbalances increase in ILP

Hardware Features to Support ILP

- Additional Functional Units
  - Advantages
    - **Does not suffer from increased latency bottleneck**
  - Disadvantages
    - **Amount of functional unit hardware proportional to degree of parallelism**
    - **Interconnection network and register file size proportional to square of number of functional units**

Hardware Features to Support ILP

- Instruction Issue Unit
  - Care must be taken not to issue an instruction if another instruction upon which it is dependent is not complete
  - Requires complex control logic in Superscalar processors
  - Virtually trivial control logic in VLIW processors
  - Big savings in power

Hardware Features to Support ILP

- Speculative Execution
  - Little ILP typically found in basic blocks
    - **a straight-line sequence of operations with no intervening control flow**
  - Multiple basic blocks must be executed in parallel
    - **Execution may continue along multiple paths before it is known which path will be executed**

Hardware Features to Support ILP

- Requirements for Speculative Execution
  - Terminate unnecessary speculative computation once the branch has been resolved
  - Undo the effects of the speculatively executed operations that should not have been executed
  - Ensure that no exceptions are reported until it is known that the excepting operation should have been executed
  - Preserve enough execution state at each speculative branch point to enable execution to resume down the correct path if the speculative execution happened to proceed down the wrong one.

Hardware Features to Support ILP

- Speculative Execution
  - Expensive in hardware
  - Alternative is to perform speculative code motion at compile time
    - **Move operations from subsequent blocks up past branch operations into proceeding blocks**
  - Requires less demanding hardware
    - **A mechanism to ensure that exceptions caused by speculatively scheduled operations are reported if and only if flow of control is such that they would have been executed in the non-speculative version of the code**
    - **Additional registers to hold the speculative execution state**
  - Not power friendly

Conclusions

- In Superscalar processors
  - architecture is “self-managed”
  - notably instruction dependence analysis and scheduling done by hardware
- In EPIC/VLIW processors
  - compiler manages hardware resources
  - synergy between compiler and architecture is key
  - some compiler optimizations will be covered in depth
  - The technology for embedded processors

Implications to Embedded Systems

- VLIW architectures are simpler designs offering the ability to reduce power requirements
- VLIW architectures allow the compiler to statically schedule instructions
  - Timing of the schedule can be controlled
    - **Real-Time Applications**
  - Power consumption can be controlled
    - **The ordering of the instructions in the schedule have power implications**
- VLIW balances power, area, and performance that makes it attractive for embedded processing