This Set

These slides do not give detailed coverage of the material. See class notes and solved problems (last page) for more information.

Text covers multiple-issue machines in Chapter 4, but does not cover most of the topics presented here.

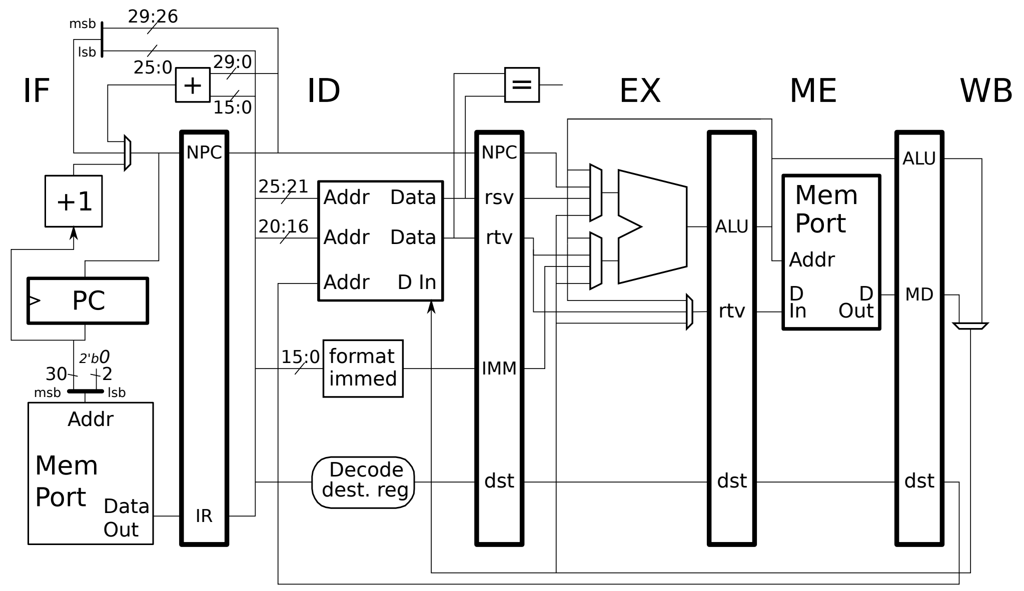
Outline

- Superscalar Machines
- VLIW Machines
- Short Vector (SIMD) Instructions
- Sample Problems

Routes to Higher Performance

We Are Here

The elegant and efficient five-stage RISC implementation.



We have the fastest device technology available (assume).

We have the most talented digital logic designers (assume).

What if our five-stage implementation is not fast enough?

Routes to Higher Performance

Faster Implementations — Higher Peak Performance

Deeply Pipelined Implementations: More stages $\therefore$ higher ϕ.

Multiple Issue, *Superscalar Implementations*: Handle > 1 insn per cycle.

Short Vector (SIMD) Instructions

Smarter Implementations — Higher Typical Performance

Dynamic Scheduling

Branch Prediction

Parallel Implementations — As much performance as you can afford*!

Multi-Core Chips, *Multiprocessors*

Computing Clusters

Distributed Systems

* Parallelization costs may apply. Results not guaranteed. Not all code is parallelizable, and not all parallelizable code is parallizable by all programmers. Code may run slower, may be more difficult to debug, and harbor more latent bugs. Parallelization can be frustrating, not responsible for broken keyboards, monitors, etc.

Multiple Issue

Multiple-Issue Machine:

A processor that can sustain fetch and execution of more than one instruction per cycle.

n-Way Superscalar Processor:

A multiple issue machine that can sustain execution of n instructions per cycle.

Scalar (Single-Issue) Processor:

A processor that can sustain execution of at most one instruction per cycle. A neologism for the five-stage MIPS implementation we have been working with.

Sustain Execution of n IPC:

Achieve a CPI of $\frac{1}{n}$ for some code fragment ...

... written by a friendly programmer ...

... to avoid cache misses and otherwise avoid stalls.

Types of Multiple Issue Machines

Superscalar Processor:

A multiple-issue machine that implements a conventional ISA (such as MIPS and SPARC).

Code need not be recompiled.

General-purpose processors were superscalar starting in early 1990's.

VLIW Processor:

A multiple-issue machine that implements a VLIW ISA ...
... in which simultaneous execution considered. (More later.)

Since VLIW ISAs are novel, code must be re-compiled.

Idea developed in early 1980's, ...
... so far used in special-purpose and stillborn commercial machines, ...
... and is being used in Intel's one-time next-generation processor.

Intel's Itanium implements the Itanium (née IA-64) VLIW ISA.

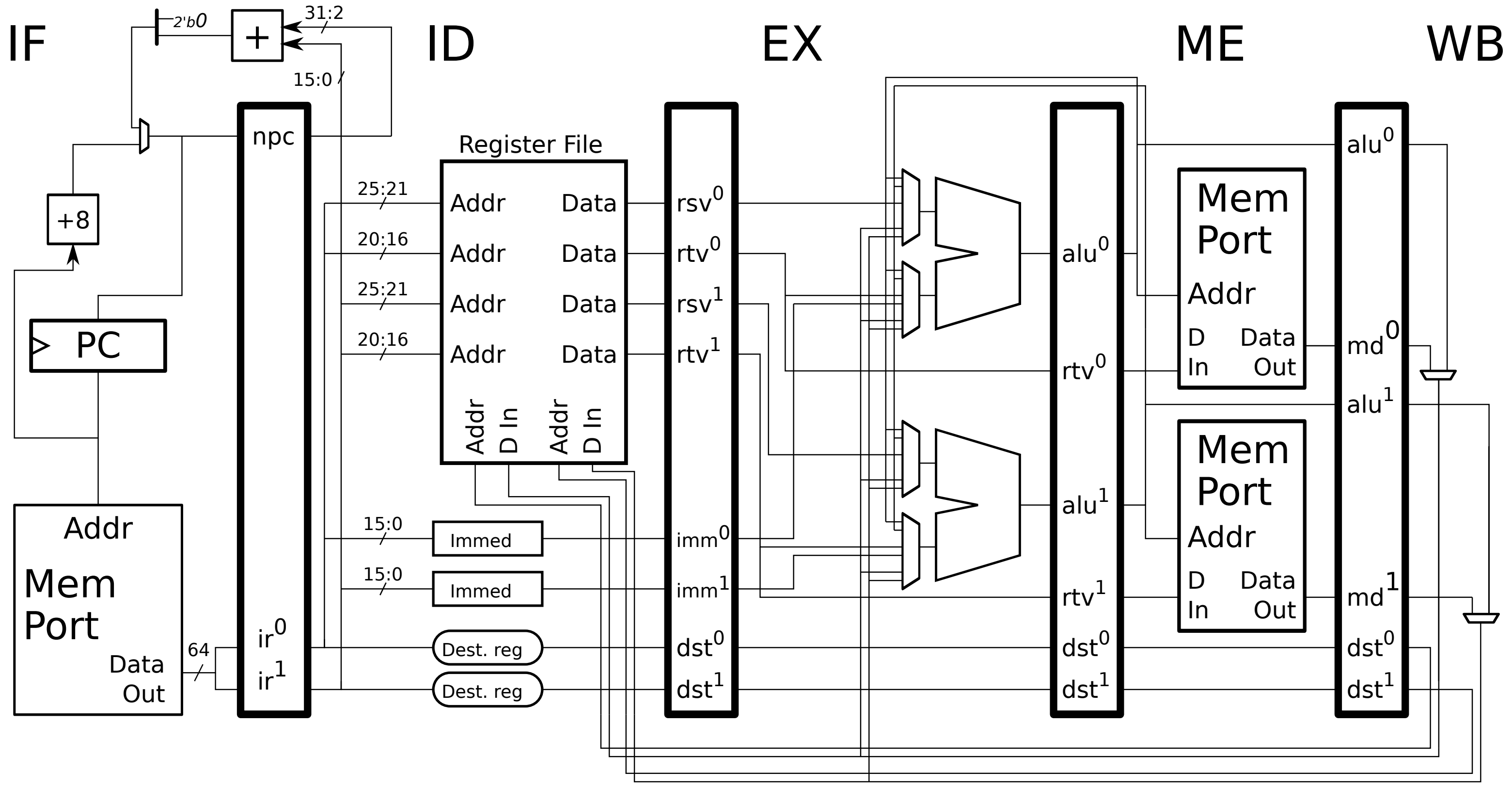
(Name of ISA and implementations are both Itanium.)

n-Way Superscalar Machine Construction

Start with a scalar, a.k.a. single-issue, machine.

Duplicate hardware so that most parts can handle n instructions per cycle.

Don't forget about control and data hazards.



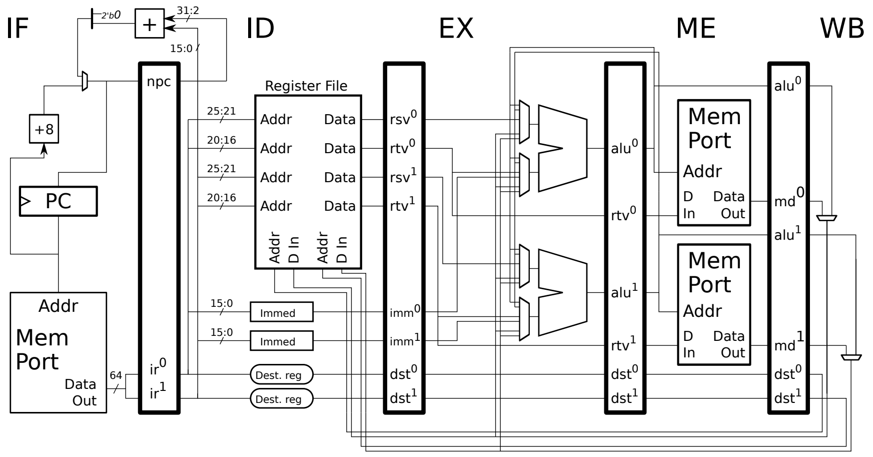
Stage-Related Terminology

Each stage has n *slots*.

Each slot holds one instruction.

Superscript on stage label indicates slot.

Superscripts often omitted for brevity.



Execution of simple code on 2-way SS MIPS:

0x1000:	# Cycle	0	1	2	3	4	5	6
add	r1, r2, r3	IF^0	ID^0	EX^0	ME^0	WB^0		
sub	r4, r5, r6	IF^1	ID^1	EX^1	ME^1	WB^1		
or	r7, r1, r8		IF^0	ID^0	EX^0	ME^0	WB^0	
and	r9, r7, r10		IF^1	ID^1	--->	EX^1	ME^1	WB^1

Note that the stall in cycle 2 would not occur on a scalar MIPS.

Stalls in Superscalar Implementations

As before, stall for true data dependencies. . .
. . . but now there are more of them.

Stall for structural hazards. (See examples further ahead.)

Stall to keep instructions in program order.

Program-order stalls make control logic much simpler.

See example on next slide.

Example of stalling to keep instructions in program order.

In the example below `xor` is stalled to keep instructions in order.

0x1000:	# Cycle	0	1	2	3	4	5	6
or	r7, r1, r8	$\texttt{IF}^0$	$\texttt{ID}^0$	$\texttt{EX}^0$	$\texttt{ME}^0$	$\texttt{WB}^0$		
and	r9, r7, r10	$\texttt{IF}^1$	$\texttt{ID}^1$	--->	$\texttt{EX}^1$	$\texttt{ME}^1$	$\texttt{WB}^1$	
xor	r2, r3, r4		$\texttt{IF}^0$	--->	$\texttt{ID}^0$	$\texttt{EX}^0$	$\texttt{ME}^0$	$\texttt{WB}^0$
add	r5, r6, r8		$\texttt{IF}^1$	--->	$\texttt{ID}^1$	$\texttt{EX}^1$	$\texttt{ME}^1$	$\texttt{WB}^1$

The `and` stalls due to a data dependence.

The `add` stalls because in cycle 2 $\texttt{ID}^1$ is occupied by `and`.

The `xor` stalls to keep the instructions in `ID` in program order:

In cycle 2: $\texttt{ID}^0$ is empty and $\texttt{ID}^1$ holds `and`.

In cycle 3: $\texttt{ID}^0$ holds `xor` and $\texttt{ID}^1$ holds `add`.

Comparison of scalar, 2-way- and 4-way-superscalar execution.

```
0x1000:  # Cycle  0    1    2    3    4    5    6    7    -- Scalar
 add r1, r2, r3   IF   ID   EX   ME   WB
 sub r4, r5, r6        IF   ID   EX   ME   WB
 or  r7, r1, r8             IF   ID   EX   ME   WB
 and r9, r7, r10                 IF   ID   EX   ME   WB

0x1000:  # Cycle  0    1    2    3    4    5    6    7    -- 2-way
 add r1, r2, r3   IF^0 ID^0 EX^0 ME^0 WB^0
 sub r4, r5, r6   IF^1 ID^1 EX^1 ME^1 WB^1
 or  r7, r1, r8        IF^0 ID^0 EX^0 ME^0 WB^0
 and r9, r7, r10       IF^1 ID^1 ---> EX^1 ME^1 WB^1

0x1000:  # Cycle  0    1    2    3    4    5    6    7    -- 4-way
 add r1, r2, r3   IF^0 ID^0 EX^0 ME^0 WB^0
 sub r4, r5, r6   IF^1 ID^1 EX^1 ME^1 WB^1
 or  r7, r1, r8   IF^2 ID^2 ---> EX^2 ME^2 WB^2
 and r9, r7, r10  IF^3 ID^3 ---------> EX^3 ME^3 WB^3
```

For this example 4-way does not help.

Superscalar Difficulties

Register File

Scalar: 2 reads, 1 write per cycle.

n-way: $2n$ reads, n writes per cycle.

Dependency Checking and Bypass Paths For ALU Instructions

Scalar, about 4 comparisons per cycle.

n-way, about $n(2(2n+n-1) = 6n^2 - 2n$ comparisons.

Loads-Use Stalls

Scalar, only following instruction would have to stall (if dependent).

n-way, up to the next $2n-1$ instructions would have to stall (if dependent).

Superscalar Difficulties

Instruction Fetch

Memory system may be limited to aligned fetches . . .

. . . for example, if branch target is `0x1114` . . .

. . . instructions starting at `0x1110` may be fetched (and the first ignored) . . .

. . . wasting fetch bandwidth.

Typical Superscalar Processor Characteristics

Instruction Fetch

Instructions fetched in *groups*, which must be aligned in some systems.

Unneeded instructions ignored.

Instruction Decode (ID)

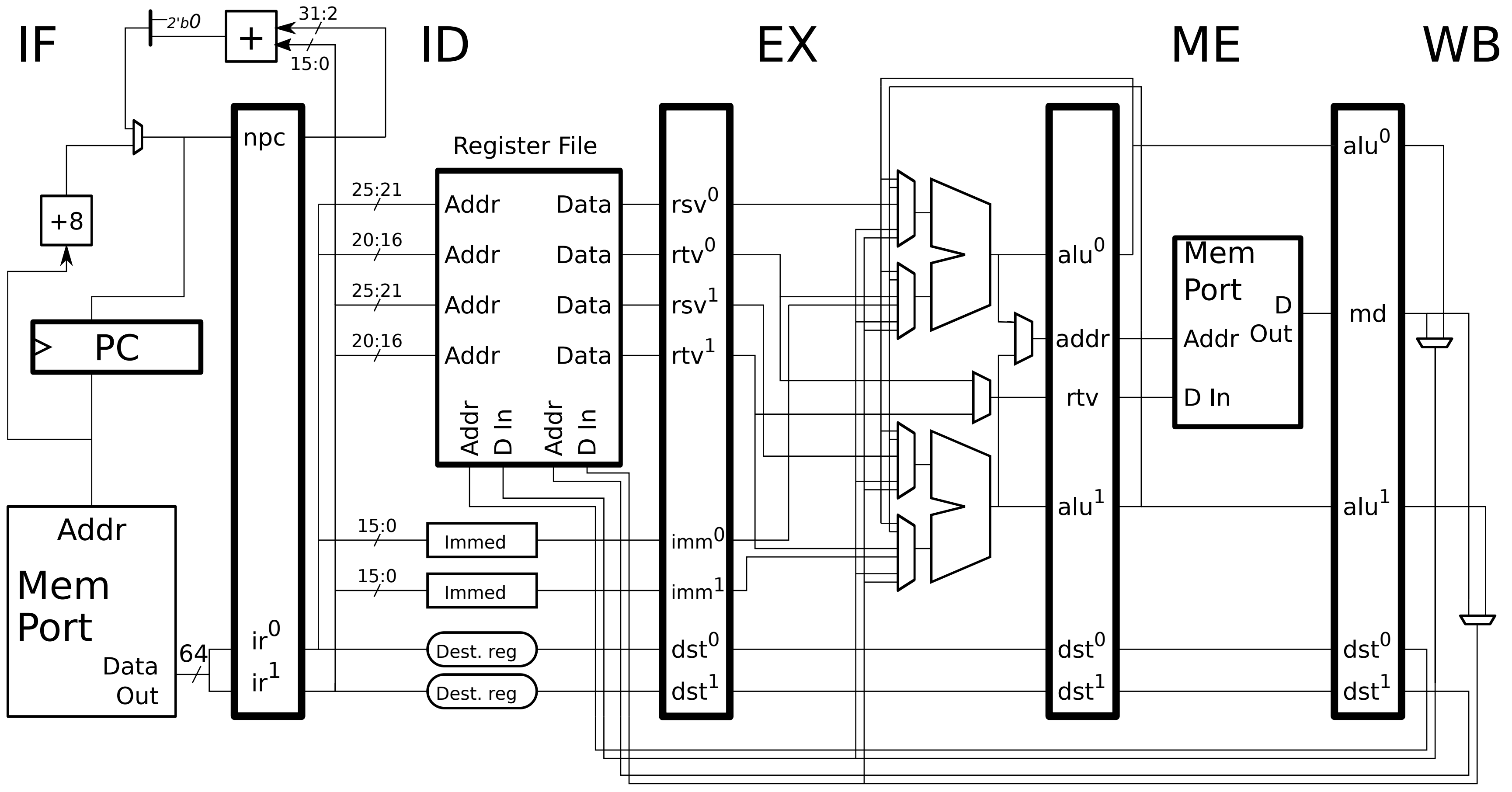
Entire group must leave ID before next group (even 1 insn) can enter.

Execution

Not all hardware is duplicated . . .
. . . and therefore some instruction pairs cause stalls.

For example, early processors could simultaneously start one floating-point and one integer instruction . . .
. . . but could not simultaneously start two integer instructions.

Example: 2-Way Superscalar MIPS with One Memory Port



VLIW

Very-Long Instruction Word (VLIW):

An ISA or processor in which instructions are grouped into *bundles* which are designed to be executed as a unit.

Explicitly Parallel Instruction Computing:

Intel's version of VLIW. Here, VLIW includes EPIC.

Key Features

Instructions grouped in bundles.

Bundles carry dependency information.

Can only branch to beginning of a bundle.

Current Examples

Texas Instruments VelociTI (Implemented in the C6000 Digital Signal Processor).

Intended for signal processors, which are usually embedded in other devices . . .
. . . and do not run general purpose code.

Intel Itanium (née IA-64) ISA (Implemented by Itanium, Itanium 2).

Intended for general purpose use.

VLIW-Related Features

Instructions grouped into 128-bit bundles.

Each bundle includes three 41-bit instructions and five *template bits*.

Template bits specify dependency between instructions and the type of instruction in each slot.

Other Features

128 64-bit General [Purpose Integer] Registers

128 82-bit FP Registers

Many additional special-purpose registers.

Makes extensive use of predication.

Cray Tera MTA implemented by the Tera Computer Company.

(Tera bought by Cray.)

Intended for scientific computing.

VLIW-Related Features

Instructions grouped into 64-bit bundles.

Each bundle holds three instructions.

Restrictions: one load/store, one ALU, and one ALU or branch.

Bundle specifies number of following non-dependent bundles in a *lookahead* field.

Serial bit for specifying intra-bundle dependencies.

Other Features

Radical: Can hold up to 128 threads, does not have data cache.

Ordinary: 32 64-bit registers.

Extra bits on memory words support inter-processor synchronization.

Branches can examine any subset of 4 condition code registers.

VLIW Bundle and Slot Definitions Definitions

Bundle: a.k.a. *packet*

The grouping of instructions and dependency information which is handled as a unit by a VLIW processor.

Slot:

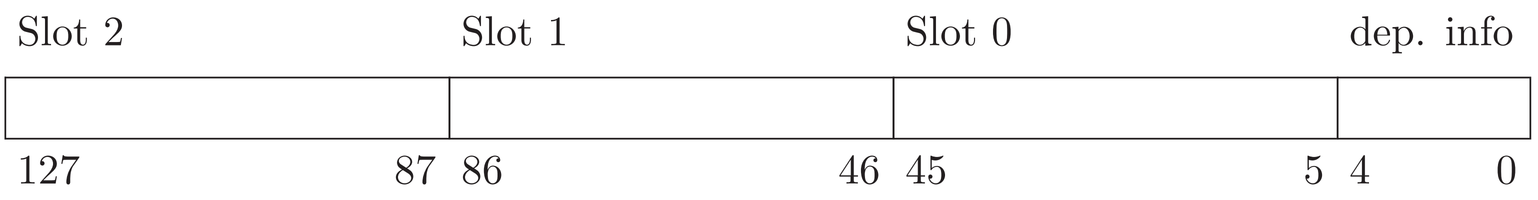
Place (bit positions) within a bundle for an instruction.

A typical VLIW ISA fits three instructions into a 128-bit bundle . . .

. . . such a bundle is said to have three slots.

Example: Itanium (née IA-64)

Bundle Size, 128 bits; holds three instructions.



Instruction Restrictions In Bundles

ISA may forbid certain instructions in certain slots . . .
. . . *e.g.*, no load/store instruction in Slot 1.

Tera-MTA: Three slots per 64-bit bundle. (Slot 0, Slot 1, Slot 2.)

Slot 0: Load/Store

Slot 1: ALU

Slot 2: ALU or Branch

Itanium (née IA-64): Three slots per 128-bit bundle.

Slot 0: Integer, memory or branch.

Slot 1: Any instruction

Slot 2: Any instruction that doesn't access memory.

There are further restrictions.

Dependency Information in Bundles

Common feature: Specify boundary between dependent instructions.

```
add r1, r2, r3
sub r4, r5, r6
! Boundary: because of r1 instruction below might wait.
xor r7, r1, r8
```

Because dependency information is in bundle less hardware is needed to detect dependencies.

How Dependency Information Can Be Specified (Varies by ISA):

- *Lookahead:*
  Number of bundles before the next true dependency.

- *Stop:*
  Next instruction depends on earlier instruction.

- *Serial Bit:*
  If 0, no dependencies within bundle(can safely execute in any order).

Specifying Dependencies Using Lookahead

Used in: Tera MTA.

Lookahead:

The number of consecutive following bundles not dependent on current bundle.

If lookahead 0, may be dependencies between current and next bundle.

If lookahead 1, no dependencies between current and next bundle, but may be dependencies between current and 2nd following bundle.

Setting the lookahead value:

Compiler analyzes dependencies in code, taking branches into account.

Sets lookahead based on nearest possible dependency.

Lookahead Example: (Two-instruction bundles.)

```
Bundle1: add r1, r2, r3
         add r4, r5, r6
         Lookahead = 1   ! Bundle 2 not dependent.

Bundle2: add r7, r7, r9
         add r10, r11, r12
         Lookahead = 2   ! Bundle 3 and Bundle 1 not dependent.

Bundle3: add r2, r1, r14
         bneq r20, Bundle1
         Lookahead = 0   ! Bundle 1 is dependent.

Bundle4: add r18, r8, r19
         bneq r21, Bundle1
         Lookahead = 11  ! Assuming twelfth bundle below uses r18.

Bundle5: nop
         nop

! (Next 10 bundles contain only nops)
```

Specifying Dependencies Using Stops

Used by: Itanium (née IA-64)

Stop:
Boundary between instructions with true dependencies and output dependencies.

Stop (and taken branches) divide instructions into *groups*.

Groups can span multiple bundles.

Within a group true and output register dependencies are not allowed, with minor exceptions.

Memory dependencies are allowed.

Assembler Notation (Itanium): Two consecutive semicolons: `;;`.

Example:

```
L1: add r1= r2, r3
L2: add r4= r5, r6 ;;
L3: add r7= r1, r0 ;;
L4: add r8= r7, r0
L5: add r9= r4, r0
! Three groups: Group 1: L1, L2;   Group 2: L3;   Group 3: L4, L5
```

VLIW and Superscalar Comparison

What is Being Compared

An n-way superscalar implementation of conventional ISA.

An n-way implementation of a VLIW ISA.

Common Benefit

Can potentially execute n instructions per cycle.

Vector Instructions

SW Idea:

CPU has a set of *vector registers*, typically 128 to 512 bits.

Each register holds several values.

Vector instruction performs operation on each value.

Example: (Intel-64 AVX)

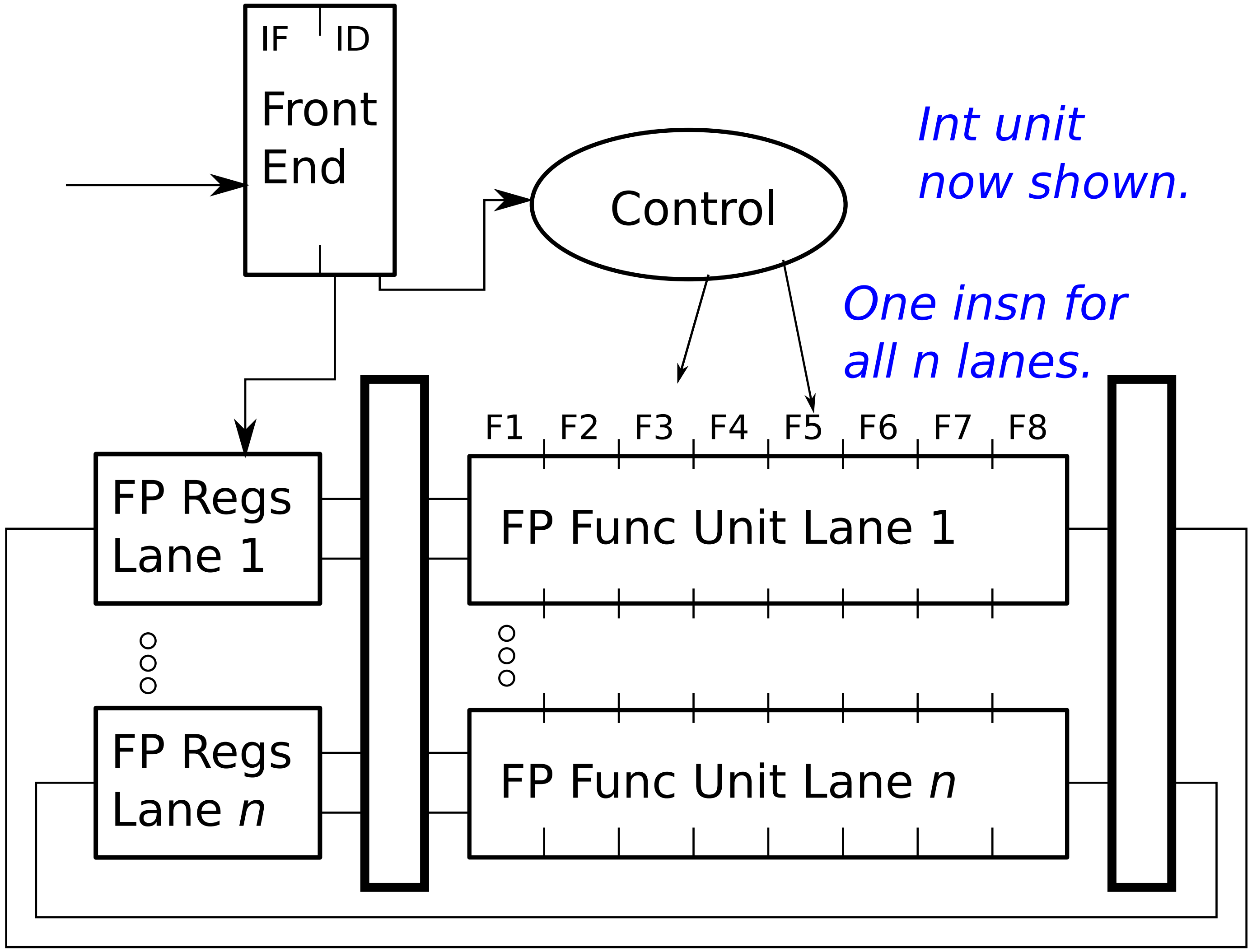
Let `ymm0` - `ymm15` be 256-bit vector registers, each holding 8 singles.

```
#  ymm9 = { 1.1,  1.2,  ..., 1.8 }
#  ymm8 = { 2.01, 2.02, ..., 2.08 }

vaddps    %ymm9, %ymm8, %ymm10    # ymm10 = ymm9 + ymm8

# ymm10 = {3.11, 3.22, ... 3.88}.
```

Equivalent MIPS Code

```
add.s f0, f2, f4
add.s f6, f8, f10
add.s f12, f14, f16
add.s f18, f20, f22

add.s f24, f26, f28
add.s f30, f32, f34
add.s f36, f38, f40
add.s f42, f44, f46
```

Vector Instruction Implementation



Vector Instruction ISA Extensions

IA-32, Intel 64

First Vector Extension: *MMX*— 64-bit vector registers.

SSE, SSE2-SSE4: 128-bit vector registers.

AVX, AVX2: 256-bit vector registers.

AVX512: 512-bit vector registers.

ARM:

A64 *Advanced SIMD*: 32×128-bit vector registers.

A32, T16 *Advanced SIMD*: 32×64-bit vector registers.

Deep Pipelining

Deep Pipelining:
Increasing or using a large number of stages to improve the performance.

If each stage in a base design can be divided into exactly n stages . . .

. . . such that the critical path in the new stages is $\frac{1}{n}$ of the base design . . .

. . . and if pipeline latches have zero setup time . . .

. . . then performance will be n times larger.

Pipelining Performance

Let t_n denote the time or an instruction to traverse an n-stage pipe.

Let t_L denote the setup time for a pipeline latch.

The latency of an n-stage unit is then

$$t_n = t_1 + (n-1)t_L$$

and the clock frequency is

$$\phi = \left(t_L + \frac{t_1}{n}\right)^{-1}; \qquad \text{or when } t_L \ll \frac{t_1}{n}, \qquad \phi \approx \frac{n}{t_1},$$

assuming that the unit is split perfectly into n pieces.

Parallelism

Parallelism:

Execution of multiple operations at the same time.

Serial Execution Model:

An execution model in which instructions have an exact program-determined order in which an instruction starts only after its predecessor finishes.

Instruction-Level Parallelism:

The parallel execution of instructions of a program in a serial execution model such that results are no different than if the instructions executed serially.