

## GPU Microarchitecture Note Set 2—Cores

- Quick Assembly Language Review
- Pipelined Floating-Point Functional Unit (FP FU)
- Typical CPU Statically Scheduled Scalar Core
- Typical CPU Statically Superscalar Core
- Bypass Network (Brief Mention)

Goal: Maximize operation throughput of a chip.

Approach:

We start with a target area (or number of gates) and power budget.

Chip will consist of multiple cores.

Find a core design that maximizes...

... FLOPS per unit area ...

... or FLOPS per unit power.

Then fill chip.

Do this with a *target workload* in mind.

## Assembly Language Examples.

### Arithmetic Instructions:

#### # Integer Instructions

```
mul r1, r2, r3    # r1 = r2 * r3
add r4, r4, r1     # r4 = r4 + r1
```

#### # Floating-Point Instructions

```
mul.s f1, f2, f3   # f1 = f2 * f3
add.s f4, f4, f1    # f4 = f4 + f1
```

## Assembly Language Examples.

### Loads and Stores

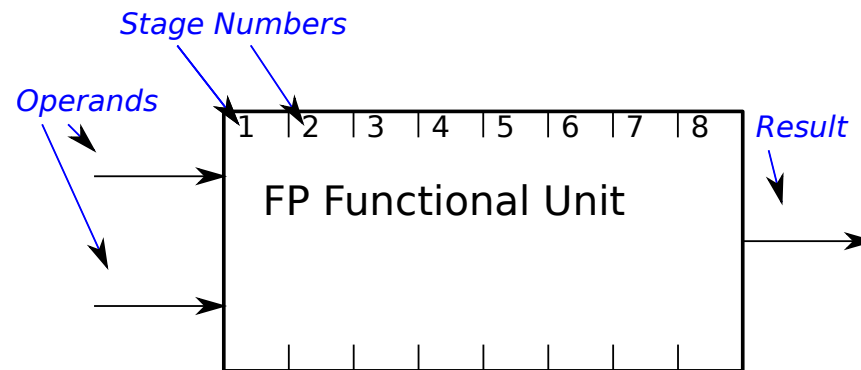
```
ld r1, [r20]    # Load: r1 = Mem[r20].    Register r20 holds a memory addr.  
ld r2, [r20+4]  # Load: r1 = Mem[r20+4]. Use of offset.  
add r3, r1, r2  
st r3, [r22]    # Store: Mem[r22] = r3.
```

## Pipelined Floating-Point Functional Unit

Performs floating-point arithmetic operations.

It has two inputs for **source operands** ...  
... and an output for the **result**.

Divided into *stages*.

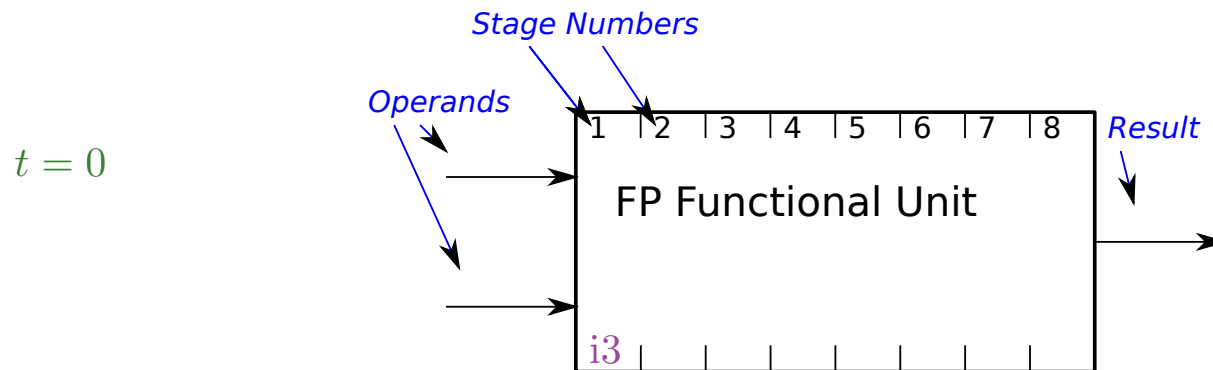


## Pipelined Operation

Operation performed in multiple **steps** ...  
 ... each step performed by a **stage**.

Illustrated unit has eight stages.

An operations starts in stage 1 ...  
 ... after 1 **clock cycle** it moves to stage 2...  
 ... *et cetera*, until it leaves stage 8 with the operation result.



i3 : add.s r1, r2, r3      # Note: r2 = 300, r3 = 3      Computes r1 = r2 + r3

## Pipelined Operation

Operation performed in multiple **steps** ...

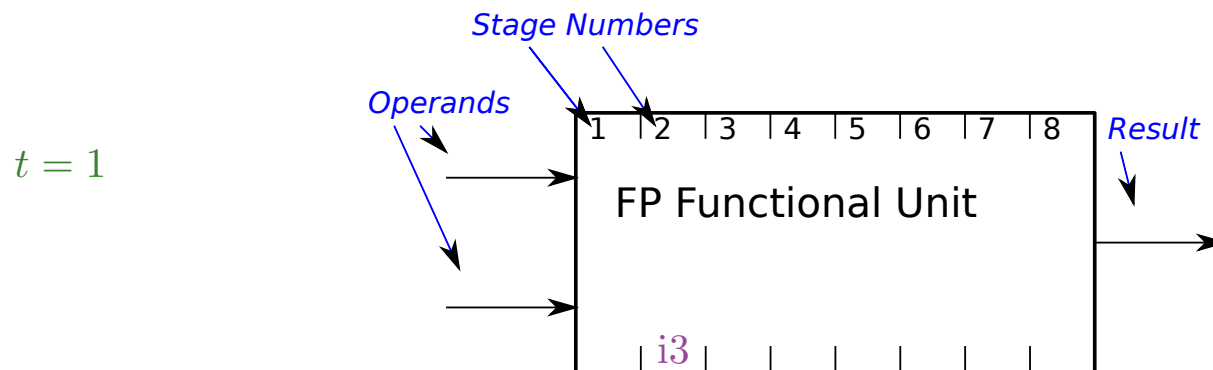
... each step performed by a **stage**.

Illustrated unit has eight stages.

An operations starts in stage 1 ...

... after 1 **clock cycle** it moves to stage 2...

... *et cetera*, until it leaves stage 8 with the operation result.



i3 : add.s r1, r2, r3      # Note: r2 = 300, r3 = 3      Computes r1 = r2 + r3

## Pipelined Operation

Operation performed in multiple **steps** ...

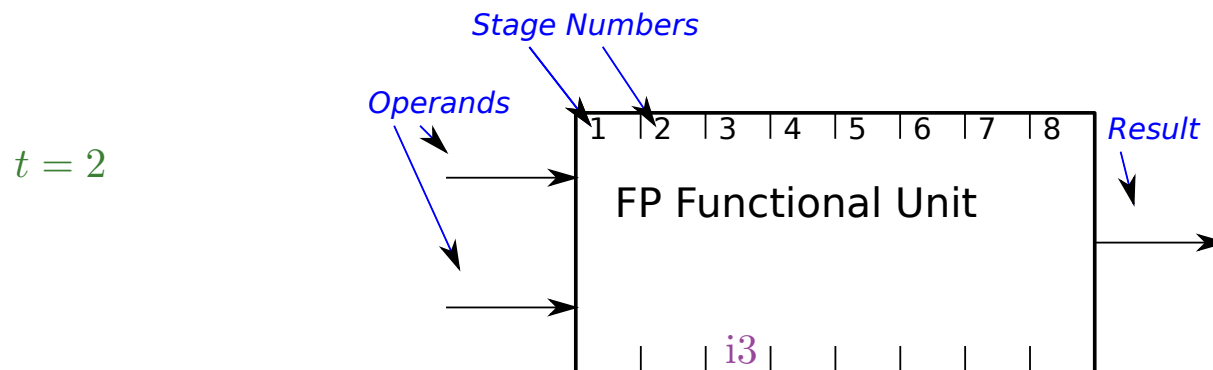
... each step performed by a **stage**.

Illustrated unit has eight stages.

An operations starts in stage 1 ...

... after 1 **clock cycle** it moves to stage 2...

... *et cetera*, until it leaves stage 8 with the operation result.



i3 : add.s r1, r2, r3      # Note: r2 = 300, r3 = 3      Computes r1 = r2 + r3



## Pipelined Operation

Operation performed in multiple **steps** ...

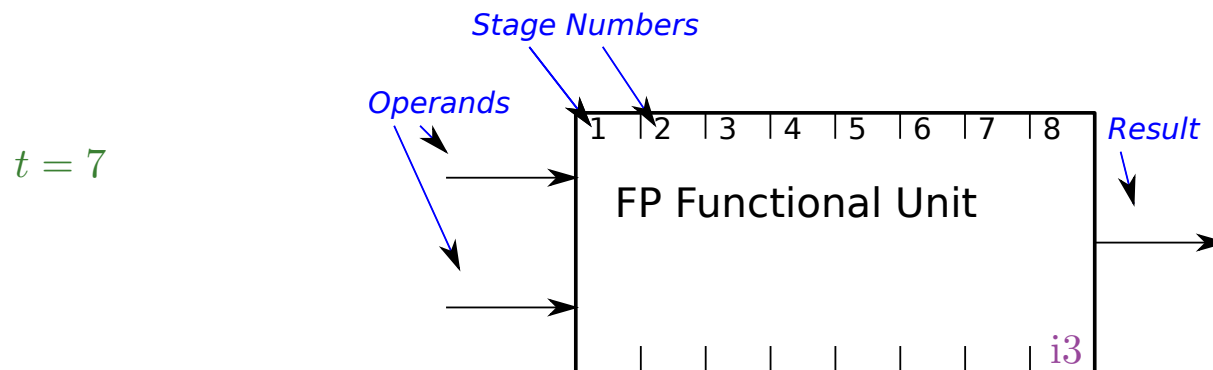
... each step performed by a **stage**.

Illustrated unit has eight stages.

An operations starts in stage 1 ...

... after 1 **clock cycle** it moves to stage 2...

... *et cetera*, until it leaves stage 8 with the operation result.



i3 : add.s r1, r2, r3      # Note: r2 = 300, r3 = 3      Computes r1 = r2 + r3

## Pipelined Operation

Operation performed in multiple **steps** ...

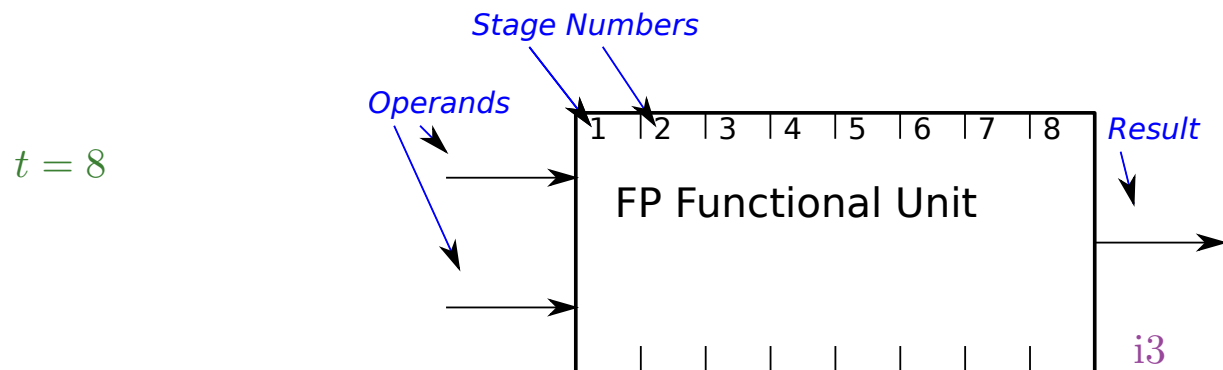
... each step performed by a **stage**.

Illustrated unit has eight stages.

An operations starts in stage 1 ...

... after 1 **clock cycle** it moves to stage 2...

... *et cetera*, until it leaves stage 8 with the operation result.



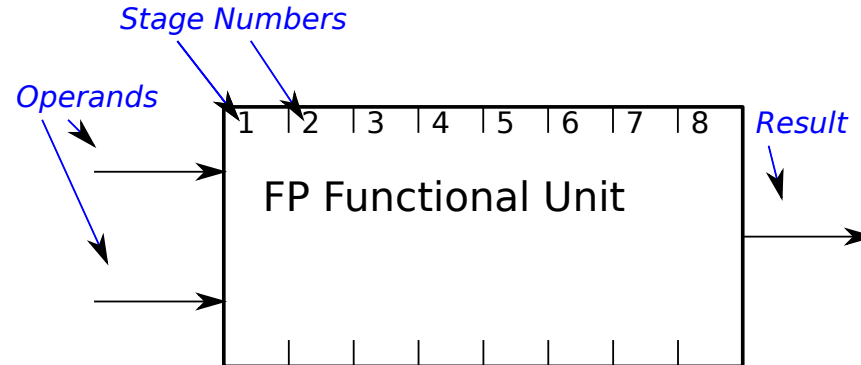
**i3** :    `add.s r1, r2, r3`        # Note:    `r2 = 300, r3 = 3`        Computes `r1 = r2 + r3`

## Pipelined Operation

Note that operation took eight cycles to perform.

The *pipeline latency* of the FP unit is 8 cycles.

(Later we'll learn that the latency of a FP instruction may be longer.)



## Pipelined Operation

To **fully utilize** the pipeline...

... we need to have **all stages occupied**.

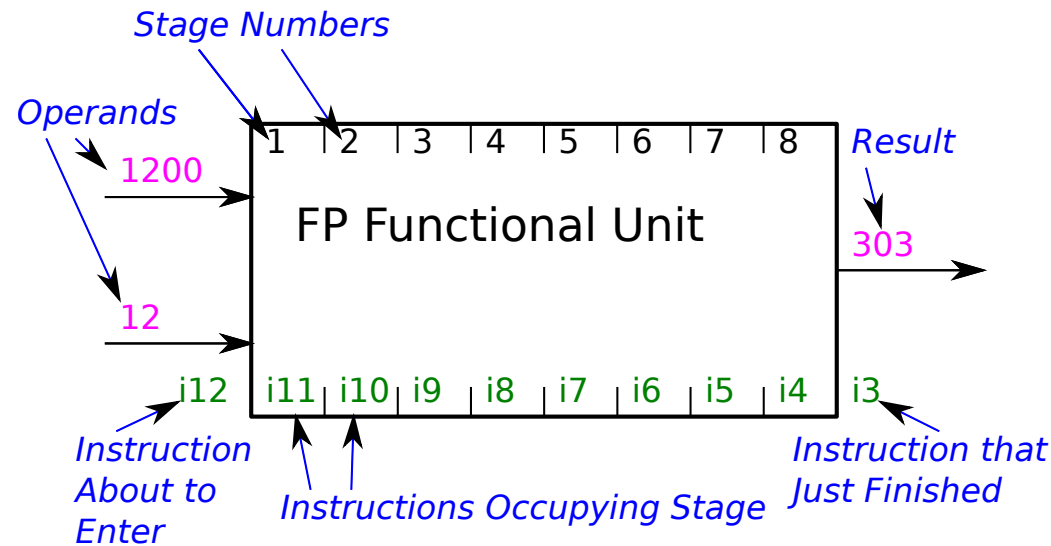
The prior example didn't fully utilize pipeline.

To fully utilize it we need an example with more instructions:

```
i3 :  add.s r1,  r2,  r3    # Note: r2 = 300,  r3 = 3      Computes r1 = r2 + r3
i4 :  add.s r4,  r5,  r6    # Note: r5 = 400,  r6 = 4      Computes r4 = r5 + r6
i5 :  add.s r7,  r8,  r9
i6 :  add.s r10, r11, r12
..
i11:  add.s r25, r26, r27
i12:  add.s r28, r29, r30   # Note: r29 = 1200,  r30 = 12
```

## Pipelined Operation

FP Pipeline Being *Fully Utilized*



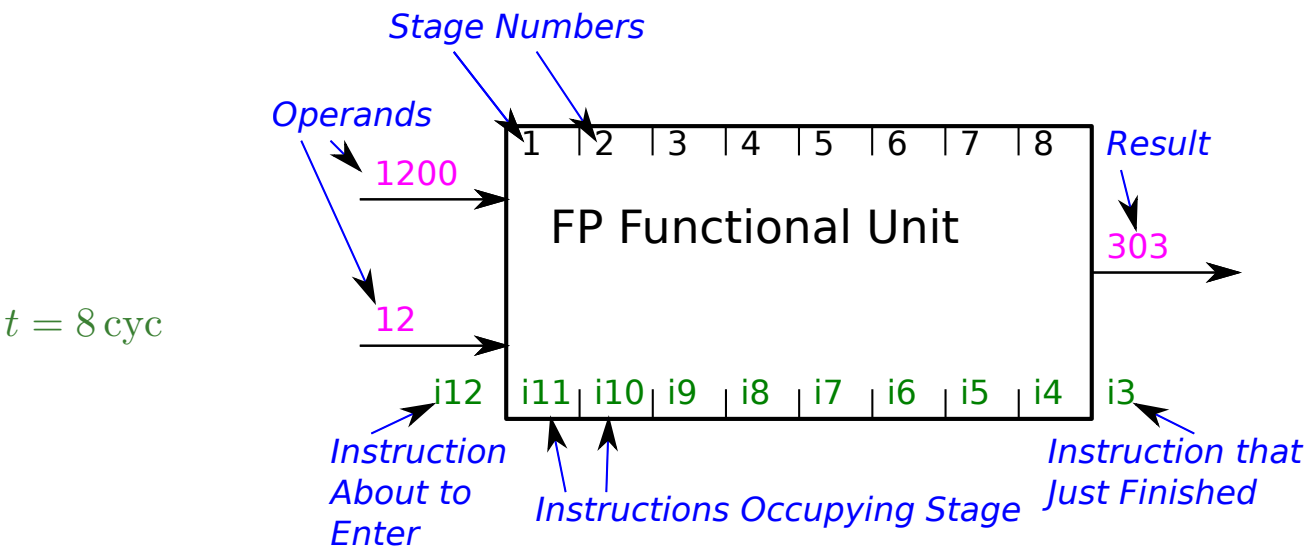
```

i3 :  add.s r1,  r2,  r3    # Note: r2 = 300,  r3 = 3      Computes r1 = r2 + r3
i4 :  add.s r4,  r5,  r6    # Note: r5 = 400,  r6 = 4      Computes r4 = r5 + r6
i5 :  add.s r7,  r8,  r9
i6 :  add.s r10, r11, r12
..
i11:  add.s r25, r26, r27
i12:  add.s r28, r29, r30   # Note: r29 = 1200,  r30 = 12

```

Pipelined Operation

FP Pipeline Being Fully Utilized



Time / Cyc -->	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
i3 : add.s r1, r2, r3	F1	F2	F3	F4	F5	F6	F7	F8									
i4 : add.s r4, r5, r6		F1	F2	F3	F4	F5	F6	F7	F8								
i5 : add.s r7, r8, r9			F1	F2	F3	F4	F5	F6	F7	F8							
i6 : add.s r10, r11, r12				F1	F2	F3	F4	F5	F6	F7	F8						
..																	
i11: add.s r25, r26, r27									F1	F2	F3	F4	F5	F6	F7	F8	
i12: add.s r28, r29, r30										F1	F2	F3	F4	F5	F6	F7	F8
Time / Cyc -->	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

## Operation Bandwidth of Pipelined FP Unit

Notice that FP unit produced a result **each** clock cycle.

Its **operation bandwidth** is 1 FLOP per cycle or  $\phi$  FLOPS ...  
... where  $\phi$  is the clock frequency.

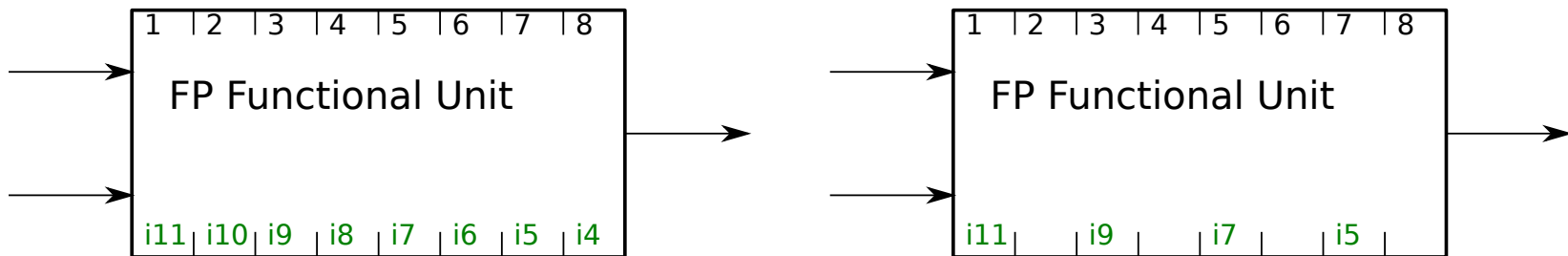
(If the clock were 4 MHz then the capability would be 4 MFLOPS.)

## Operation Bandwidth and Operation Throughput

Recall: *Bandwidth* is best that hardware can do...

... *Throughput* is what you get.

Consider Two Possible Situations:



Assume that the patterns above persist over time.

The unit on the left is fully utilized.

Bandwidth is 1 FLOP per cycle, throughput is 1 FLOP per cycle.

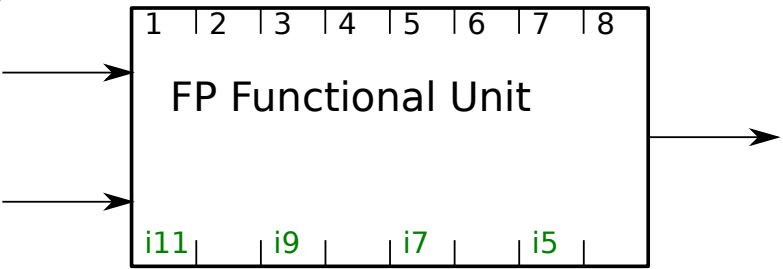
The unit on the right is half utilized.

Bandwidth is 1 FLOP per cycle, throughput is 0.5 FLOP per cycle.



Correspondence between pipeline execution diagram (text) and illustration.

Illustration below for  $t = 8$ .



Time / Cyc -->	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
i3 : add.s f1, f2, f3	F1	F2	F3	F4	F5	F6	F7	F8									
i4 : xor r4, r5, r6		EX															
i5 : add.s f7, f8, f9			F1	F2	F3	F4	F5	F6	F7	F8							
i6 : xor r10, r11, r12				EX													
i7 : add.s f10, f11, f12					F1	F2	F3	F4	F5	F6	F7	F8					
i8 : xor r13, r14, r15						EX											
..																	
i11: add.s f25, f26, f27									F1	F2	F3	F4	F5	F6	F7	F8	
i12: xor r28, r29, r30										EX							
Time / Cyc -->	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

### Why FP Unit Might **Not** be Fully Utilized

- Not every instruction uses the FP unit.
- The operand of a FP instruction is not ready, so it must wait.
- The FP instruction itself is late to arrive.

As a result of these situations ...

... throughput will be less than capability.

Non-Full Utilization Due to Non-FP Instructions

*The instructions with even numbers don't use FP unit.*

Note: EX indicates the integer operation stage (just 1 stage needed).

If pattern persists, throughput is 0.5 FLOP per cycle.

Time / Cyc -->	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
i3 : add.s f1, f2, f3	F1	F2	F3	F4	F5	F6	F7	F8									
i4 : xor r4, r5, r6		EX															
i5 : add.s f7, f8, f9			F1	F2	F3	F4	F5	F6	F7	F8							
i6 : xor r10, r11, r12				EX													
..																	
i11: add.s f25, f26, f27									F1	F2	F3	F4	F5	F6	F7	F8	
i12: xor r28, r29, r30										EX							
Time / Cyc -->	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Possible targets of blame:

- The programmer, for not being able to avoid `xor` instructions.
- The problem, which has lots of unavoidable `xor` instructions.

Non-Full Utilization Due to “Late” Operands

For some reason, i4’s operands arrive three cycles late.

We will be looking at situations that cause this later in the semester.

Time / Cyc -->	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
i3 : add.s f1, f2, f3	F1	F2	F3	F4	F5	F6	F7	F8									
i4 : add.s f4, f5, f6					F1	F2	F3	F4	F5	F6	F7	F8					
i5 : add.s f7, f8, f9						F1	F2	F3	F4	F5	F6	F7	F8				
i6 : add.s f10, f11, f12							F1	F2	F3	F4	F5	F6	F7	F8			
Time / Cyc -->	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Possible targets of blame:

- The compiler, for not doing a better job scheduling instructions.
- The programmer, for sloppy coding that hindered compiler scheduling.
- The hardware, for not having enough registers to enable scheduling.

## Number of Stages, Clock Frequency, Power

### One-Stage FP Unit

Let  $t_1$  denote the latency of the 1-stage unit.

The latency is determined by the *device technology* used ...  
... and by the design of the floating-point unit.

The technology determines how fast transistors switch ...  
... which in turn is determined by time for electric charge to clear gate junctions.

Device technology is beyond the control of computer engineers.

The *logic design* of the FP unit determines the number of gates from input to output.

We will assume that this too is beyond our control.

*Therefore* for us  $t_1$  is a constant.

Construction of  $n$ -Stage Pipelined Units

An additional component, called a *pipeline latch* is inserted between stages.

Let  $t_L$  denote the time needed for this latch.

The **latency** of an  $n$ -stage unit is then

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

and the clock frequency is

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

assuming that the unit is split perfectly into  $n$  pieces.

This doesn't sound like an improvement, but keep paying attention.

## Operation Bandwidth of $n$ -Stage Units

Note that bandwidth of the 1-stage unit is  $1/t_1$  FLOPS.

For an  $n$ -stage device the bandwidth is:

$$\frac{1}{t_1/n + t_L}.$$

If  $t_L = 0$ , this reduces to  $n/t_1$  ...

... meaning we can make the capability as high as we want by choosing  $n$  ...

... and pretending that  $t_L = 0$  and the unit can be divided perfectly.

Note that by choosing  $n$  we are choosing the clock frequency.

## Number of Stages, Clock Frequency, Power

Increasing number of stages increases:

- :- ) The operation bandwidth (as discussed above).
- :- ( The area (a cost measure) (for latches and circuit changes).
- :- ( Power (which is proportional to clock frequency).

Factors that Limit Increase in  $n$

A few years ago: area and efficiency of splitting.

Now: power.



## Pipelined FP Unit Summary

An  $n$ -stage unit takes  $n$  cycles to compute 1 operation ...

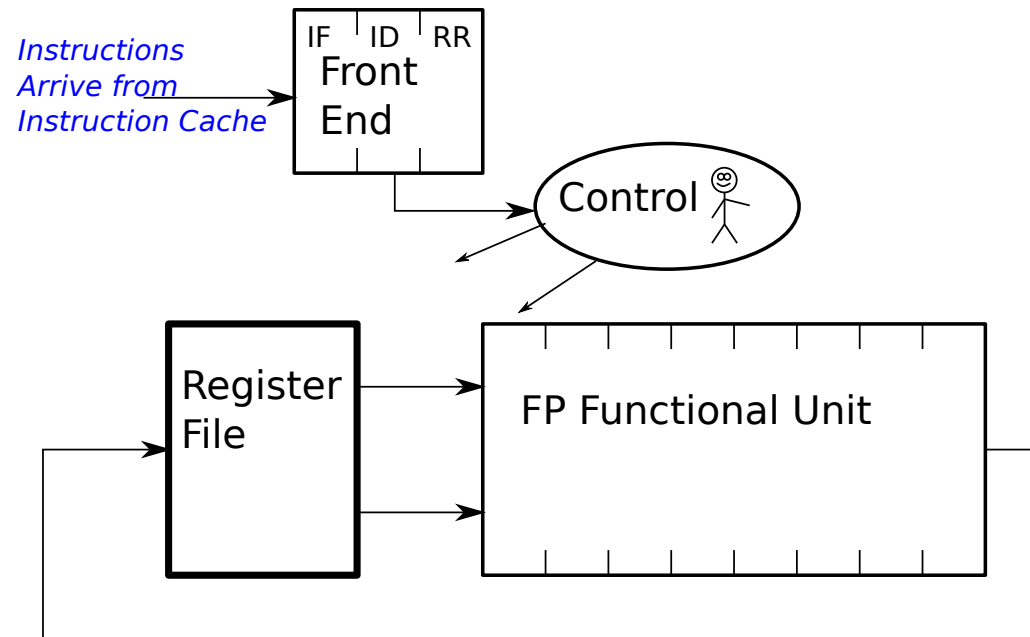
... in other words it has an  $n$ -cycle latency.

An  $n$ -stage unit can compute an operation each cycle ...

... in other words it has an operation bandwidth of 1 FLOP per cycle.

Operation throughput is limited by external factors such as instruction mix.

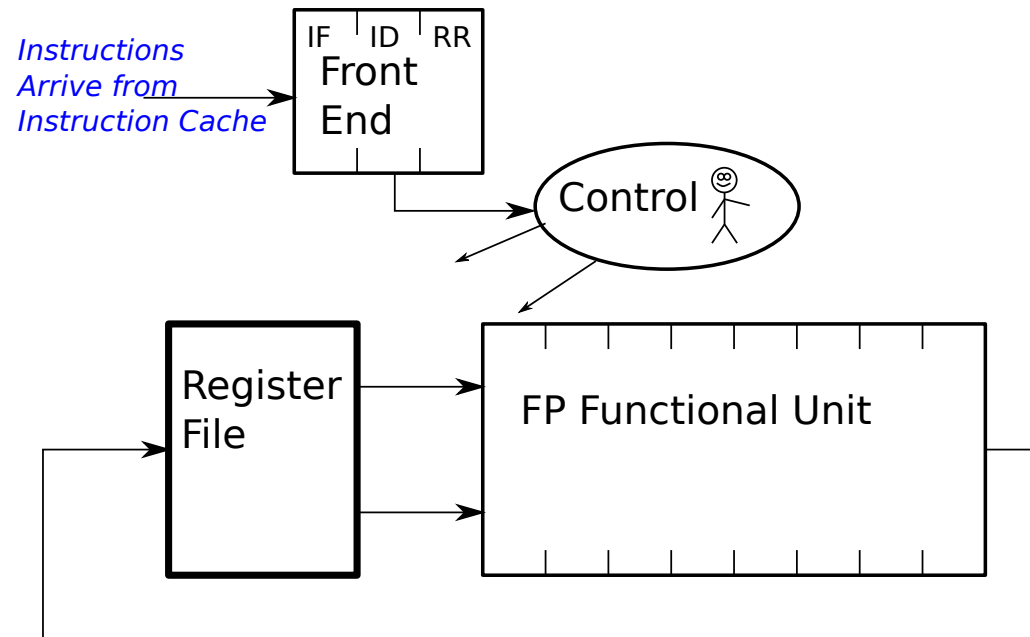
## Simple Core



## Components

- *Front End* — Hardware for fetching, decoding, and issuing instructions.
- *Register File* — A storage device for register values.
- FP Functional Unit
- *Control Unit* — Hardware sending control signals to other components.

## Simple Core

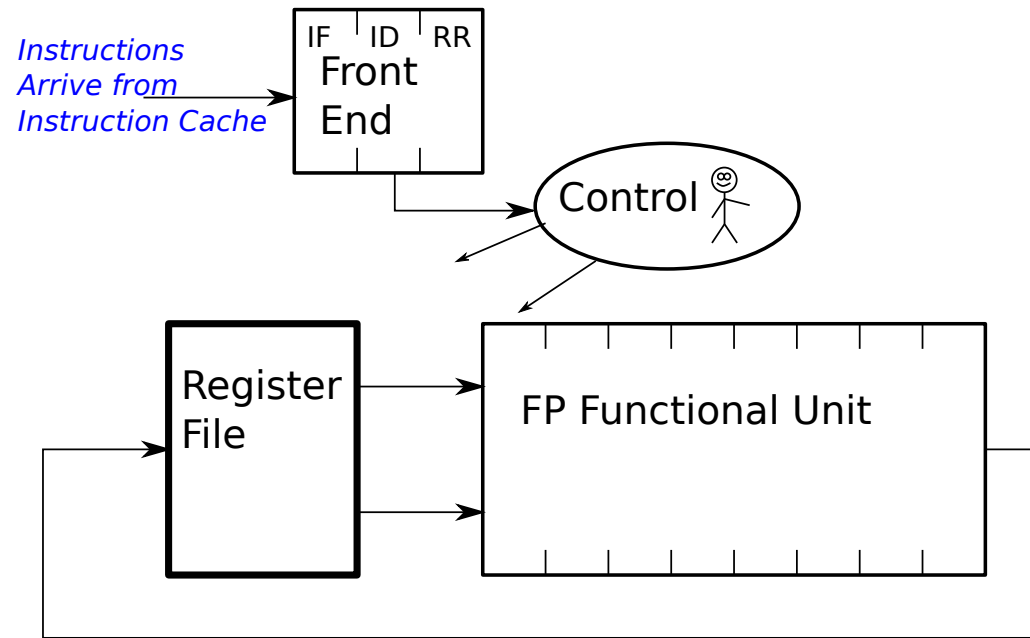
*Front End*

Pipelined, three stages shown.

Input is from instruction cache.

Stage labels: *IF*, Instruction Fetch; *ID*, Instruction Decode; *RR*, Register Read.

## Front End and Performance



The front end usually determines instruction bandwidth of core.

A core is called *scalar* if the front end can fetch  $\leq 1$  insn per cycle.

A core is called *n-way superscalar* if the front end can fetch  $n$  insns per cycle.

The simple core we are discussing is scalar.

Internal details of front end not covered in this course.

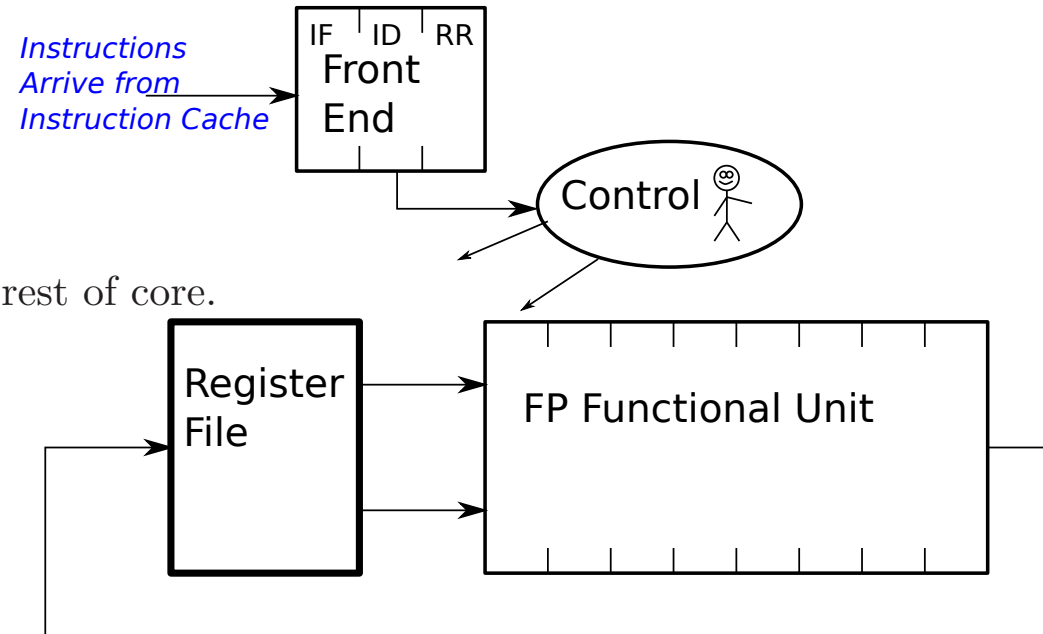
## Control Logic

## Purpose

Control logic sends commands to rest of core.

It determines when instructions must *stall* (wait).

Details omitted in this course.



## Control Logic Cost

For simple core, cost is very low.

For heavy-weight cores, cost is high.

## Integer and FP Pipelines

Simple core also has an *integer pipeline*. *Instructions Arrive from Instruction Cache*

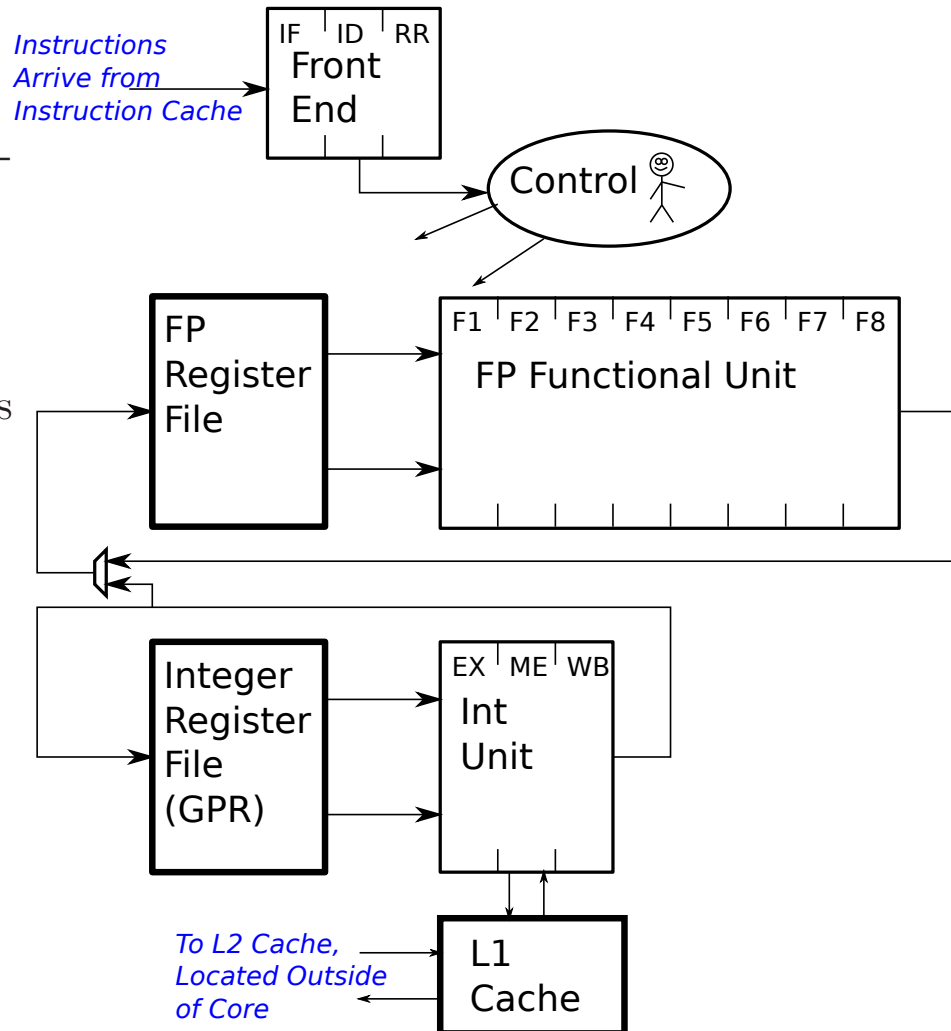
(Actually, the FP pipeline is considered optional).

Each pipeline has *its own register file*.

Integer instructions can write FP registers (needed for loads from memory).

All instructions pass through front end.

Integer instructions use integer pipeline.



## Integer and FP Pipelines

Simple core also has an *integer pipeline*. *Instructions Arrive from Instruction Cache*

**EX: Execute** — Perform the integer operation.

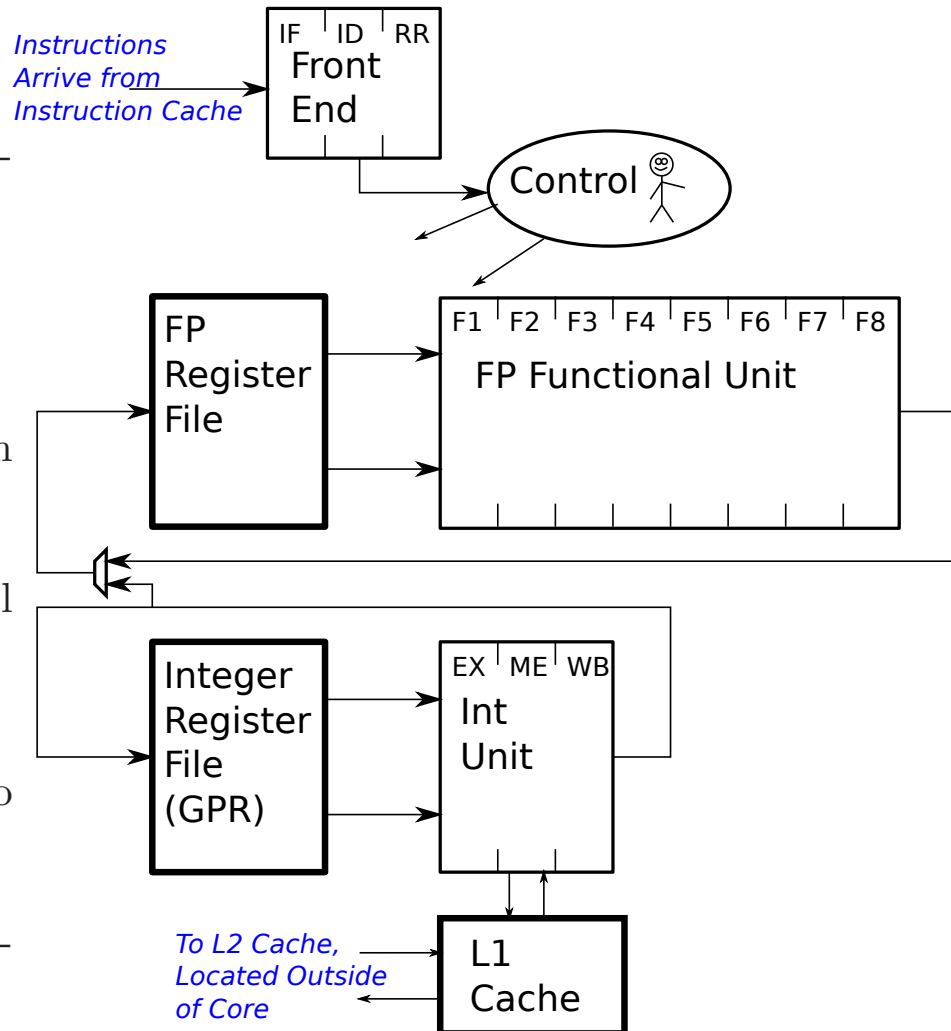
Just takes one cycle.

**ME: Memory** — Try memory operation (read or write).

In simple cores pipeline will stall until operation completes.

**WB: Writeback** — Writeback value to register file.

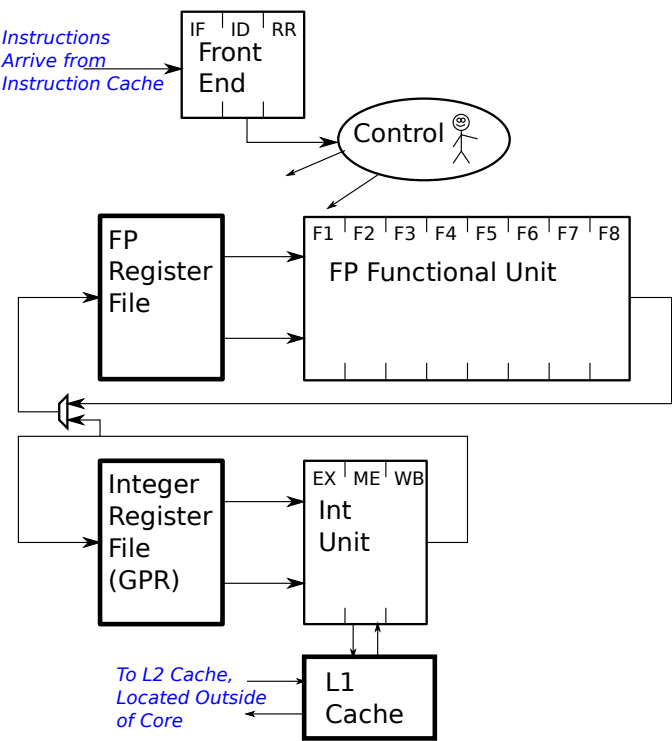
The last FP stage, **F8**, does a write-back to the FP registers.



Execution Example

Instruction throughput is 1 insn/cyc.

FP operation throughput is 0.5 op per cyc.

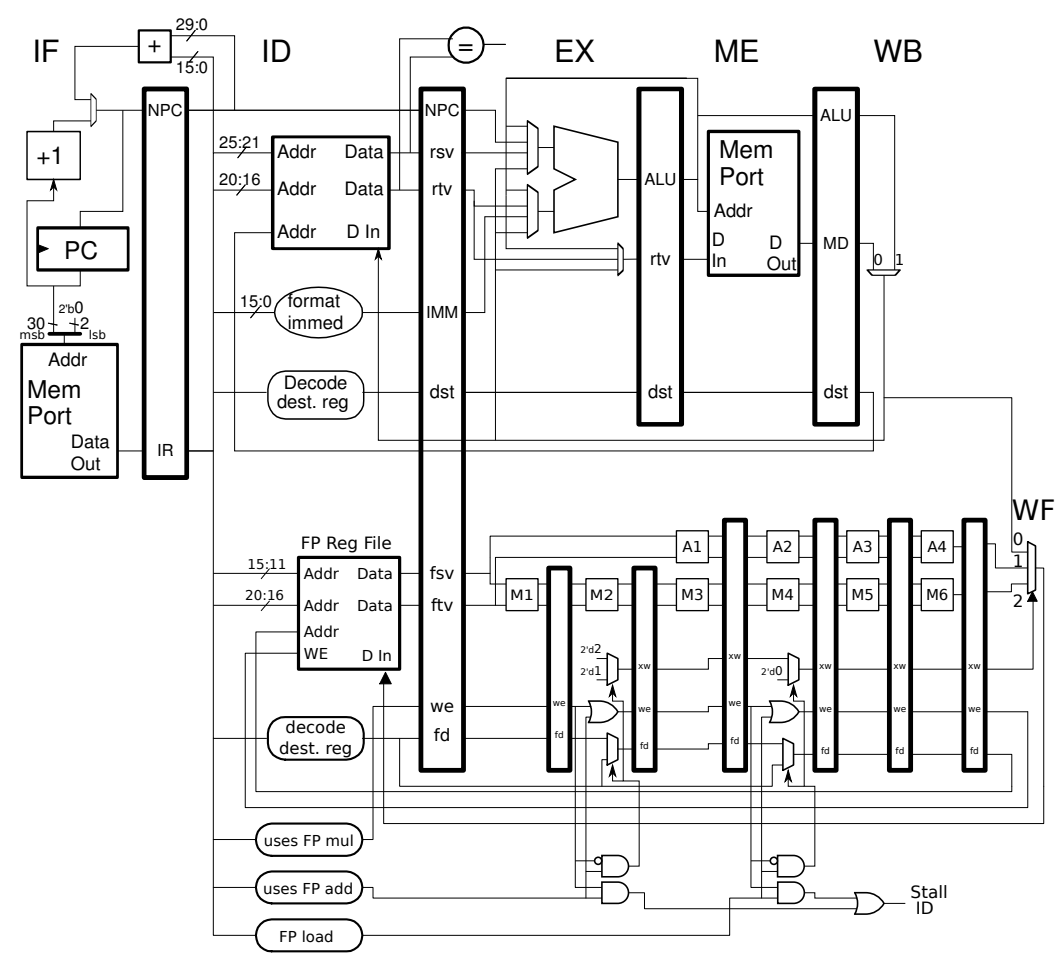


Time / Cyc -->	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15		
i3 : add.s f1, f2, f3	IF	ID	RR	F1	F2	F3	F4	F5	F6	F7	F8							
i4 : xor r4, r5, r6		IF	ID	RR	EX	ME	WB											
i5 : add.s f7, f8, f9			IF	ID	RR	F1	F2	F3	F4	F5	F6	F7	F8					
i6 : xor r10, r11, r12				IF	ID	RR	EX	ME	WB									
..																		
i11: add.s f25, f26, f27										IF	ID	RR	F1	F2	F3	F4	F5	..
i12: xor r28, r29, r30											IF	ID	RR	EX	ME	WB		
Time / Cyc -->	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15		



Integer and FP Pipelines

More detailed view, not covered in this class.



## Simple Core and Performance

First, let's use a better name than simple core:

*Statically Scheduled Scalar Core*

### Increasing Operation and Instruction Bandwidth of Simple Core

The only way to change operation and insn bandwidth...  
... is to increase the number of stages (and therefore clock frequency).

Doing so will increase FLOPS ...  
... but at some point will reduce FLOPS per unit area ...  
... and FLOPS per unit power.

The Next Step—*Superscalar*: Multiple Instructions per Cycle

## Some Definitions

### *n-Way Superscalar Core:*

A core that has an instruction bandwidth of  $n$  instructions per cycle.

FP operation bandwidth  $\leq n$ .

### *Statically Scheduled Core:*

A core in which instructions start execution in program order.

### Example: Two-Way Superscalar, 1 FP op per cycle

In this example design there are...

... **two integer units**...

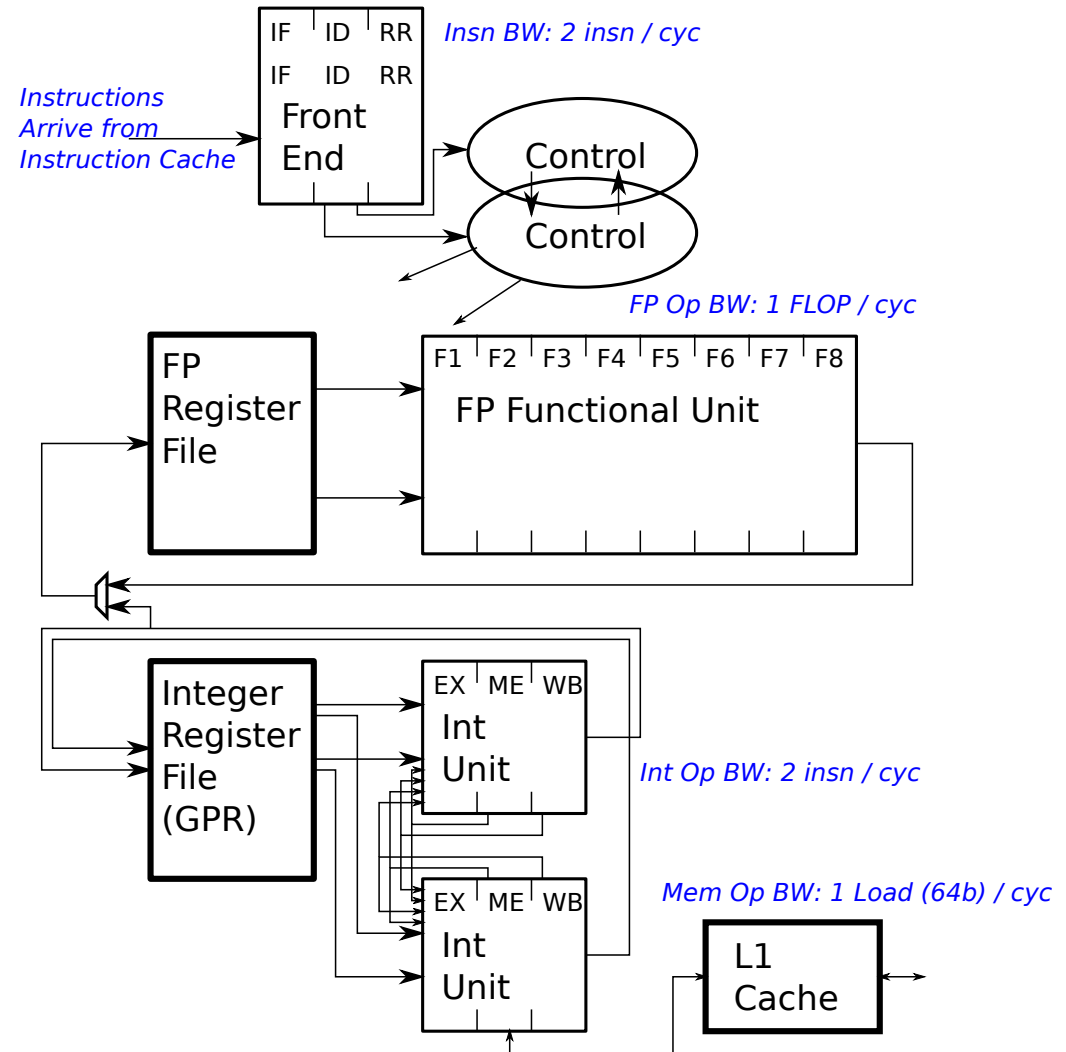
... but just **one FP unit**...

... and **one L1 data cache port**.

This organization similar to...

... 1990's general purpose CPUs...

... 2010's lightweight cores.



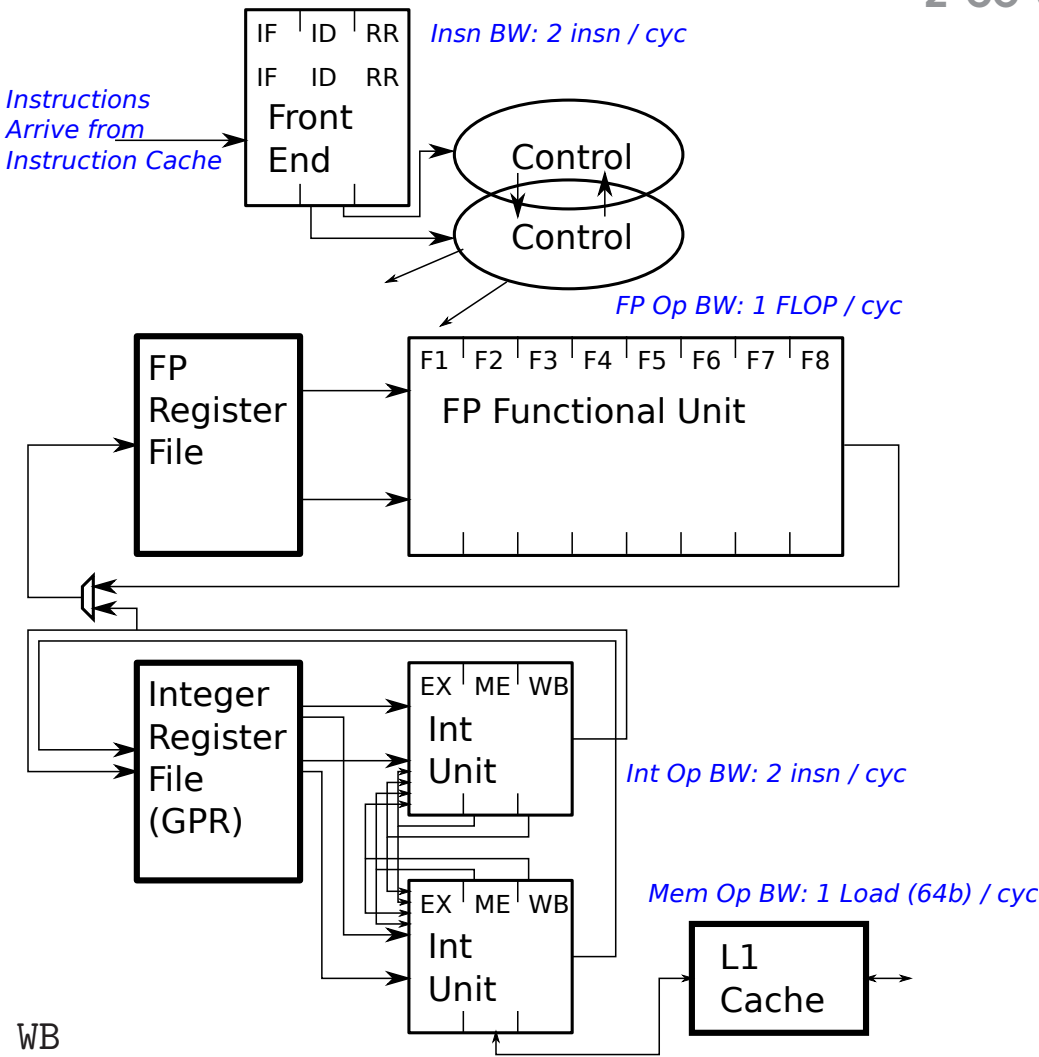
Execution Example

Note that `add` uses int unit.

For this example:

Insn BW and throughput is 2 insn/cyc.

FP operation throughput is 0.



Time / Cyc -->	0	1	2	3	4	5				
add r1, r2, r3	IF	ID	RR	EX	ME	WB				
add r4, r5, r6	IF	ID	RR	EX	ME	WB				
add r7, r8, r9		IF	ID	RR	EX	ME	WB			
add r10, r11, r12		IF	ID	RR	EX	ME	WB			
add r25, r26, r27					IF	ID	RR	EX	ME	WB
add r28, r29, r30					IF	ID	RR	EX	ME	WB
Time / Cyc -->	0	1	2	3	4	5	6	7	8	9

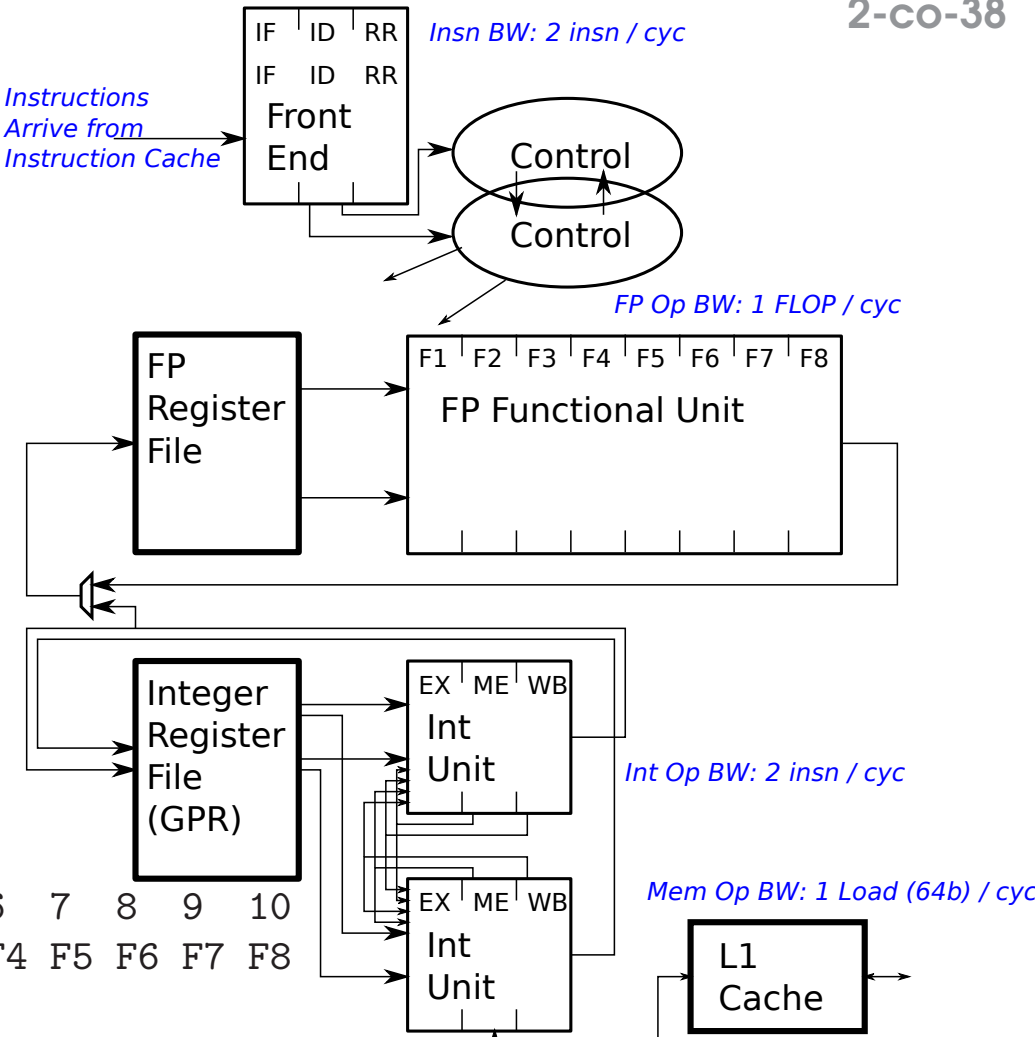
Execution Example

Note that `add.s` uses FP unit.

For this example:

Insn BW and throughput is 2 insn/cyc.

FP operation BW and throughput...  
... is 1 FLOP/cyc.



Time / Cyc -->	0	1	2	3	4	5	6	7	8	9	10								
<code>add.s f1, f2, f3</code>	IF	ID	RR	F1	F2	F3	F4	F5	F6	F7	F8								
<code>xor r4, r5, r6</code>	IF	ID	RR	EX	ME	WB													
<code>add.s f7, f8, f9</code>		IF	ID	RR	F1	F2	F3	F4	F5	F6	F7	F8							
<code>xor r10, r11, r12</code>		IF	ID	RR	EX	ME	WB												
...																			
<code>add.s f25, f26, f27</code>						IF	ID	RR	F1	F2	F3	F4	F5	..					
<code>xor r28, r29, r30</code>						IF	ID	RR	EX	ME	WB								
Time / Cyc -->	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15			

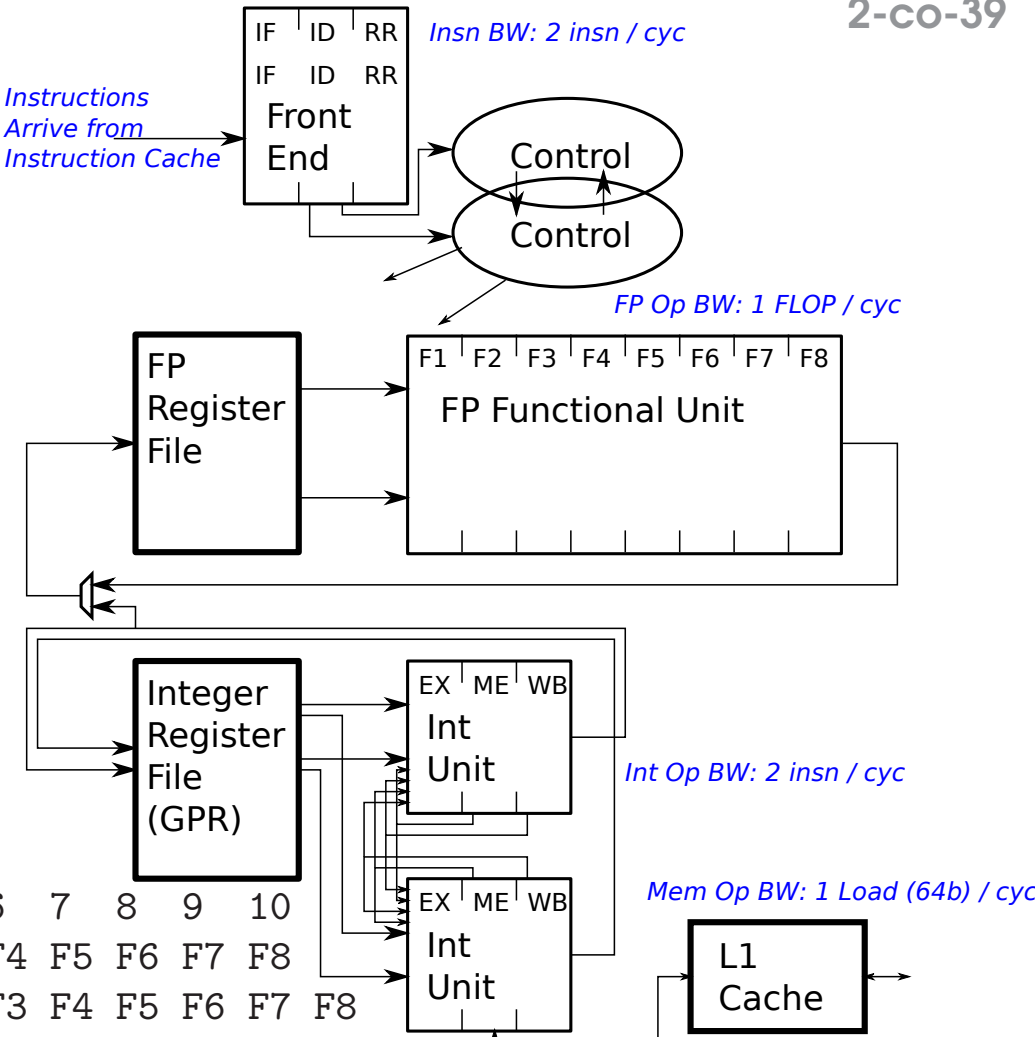
Execution Example

A -> indicates...  
... a *pipeline stall* ...  
... indicating insn and those behind it...  
... are stopped.

For this example:

Insn throughput is now 1 insn/cyc.

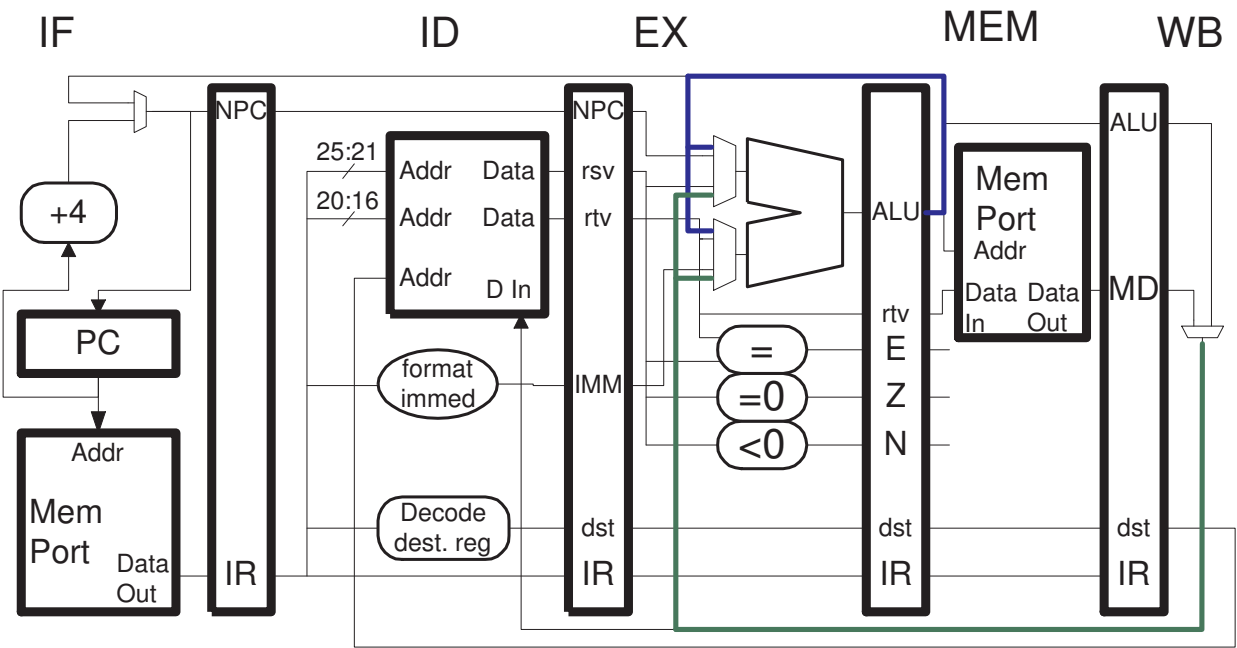
FP operation BW and throughput...  
... are 1 FLOP/cyc.



Time / Cyc -->	0	1	2	3	4	5	6	7	8	9	10											
add.s f1, f2, f3	IF	ID	RR	F1	F2	F3	F4	F5	F6	F7	F8											
add.s f4, f5, f6	IF	ID	->	RR	F1	F2	F3	F4	F5	F6	F7	F8										
add.s f7, f8, f9		IF	->	ID	RR	F1	F2	F3	F4	F5	F6	F7	F8									
add.s f10, f11, f12		IF	->	ID	->	RR	F1	F2	F3	F4	F5	F6	F7	F8								
add.s f13, f14, f15				IF	->	ID	RR	F1	F2	F3	F4	F5	F6	F7	F8							
add.s f16, f17, f18				IF	->	ID	->	RR	F1	F2	F3	F4	F5	F6	F7	F8						
...																						
Time / Cyc -->	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16					

*Bypass Network:*

Special connections from FU outputs to FU inputs that expedite the execution of dependent instructions.



Blue bypass used in cycle 3, green bypass used in cycle 4.

# Cycle	0	1	2	3	4	5	6
add r10, r2, r3	IF	ID	EX	ME	WB		
sub r4, r10, r5		IF	ID	EX	ME	WB	
xor r6, r7, r10			IF	ID	EX	ME	WB



### Bypass Network:

Special connections from FU outputs to FU inputs that expedite the execution of dependent instructions.

The output of each unit connects...

... to both inputs of each of  $n$  units ...

... and so cost of bypass network is  $O(n^2)$ .

In diagram FU output taken from **ME** and **WB** stage (because that's where the data is).

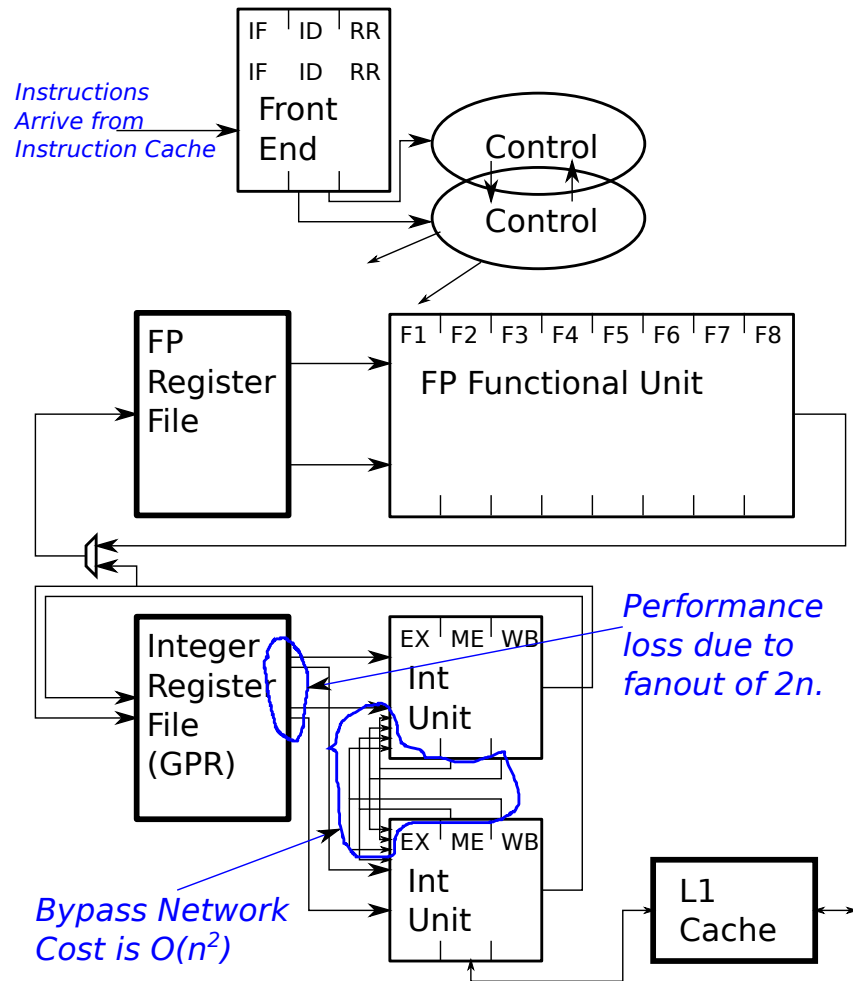
A bypass network **is expensive** ...

... because each bypass connection ...

... **is 64 bits wide** in current systems.

A bypass network **is optional** ...

... but present in most CPU cores.



## Cost and Performance Issues

$n$ -way superscalar costs:

Most items cost  $n \times$  more ...

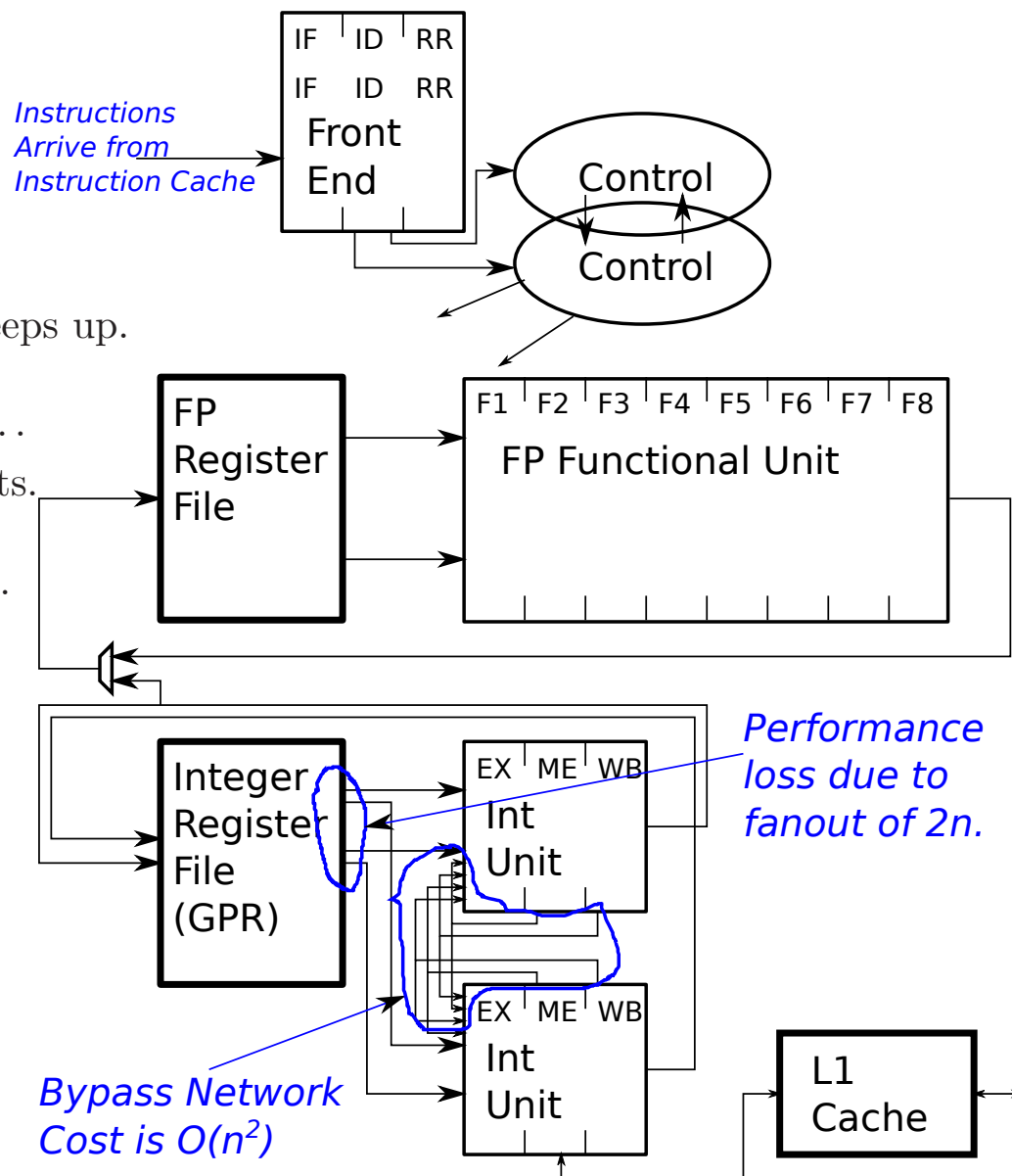
... which is good if throughput keeps up.

For some units  $< n$  would suffice...

... for example,  $n/2$  L1 cache ports.

Costliest part is *bypass network*...

... with its  $O(n^2)$  cost.



## Typical Superscalar Cores

Four-Way Superscalar

Scalar FP Bandwidth 1-2 Operations per Cycle

Vector FP Bandwidth 4-8 Operations per Cycle

## Design Limiters or Why There are No Eight-Way Superscalar Cores

Define *utilization* to be  $\text{throughput} / \text{bandwidth}$ .

Reduced front-end utilization as IB increases.

Cost of the bypass network.

*Instruction-Level Parallelism (ILP):*

The degree to which the execution of instructions in a program written in a serial ISA can be overlapped.

Most ISAs are serial, such as MIPS, Intel 64, SPARC.

Superscalar implementations overlap execution of such instructions ...

... but do so in a way that the results obtained ...

... are the same as one would get with one-at-a-time execution.

An  $n$ -way superscalar processor has a bandwidth of  $n$  insn/cyc.

The throughput of a program on the  $n$ -way processor ...

... is determined by the ILP of the program.

## Next Steps (after statically scheduled superscalar)

Further Exploitation of ILP:

- *Dynamic scheduling.*
- *Branch prediction.*

These will be defined, but not covered in detail.

For **Operation Throughput** Improvement

- Vector Instructions

For Greater **Latency Tolerance**

- *Simultaneous Multithreading (SMT)*, a.k.a. *Hyperthreading*

For **Operation Density** Improvement

- *Single-Instruction Multiple Thread (SIMT)*

## Superscalar Microarchitecture Features

### *Dynamic Scheduling* (A.k.a. *out-of-order execution*)

An organization in which instructions execute when their operands are ready, not necessarily in program order.

Consumes a lot of power and area.

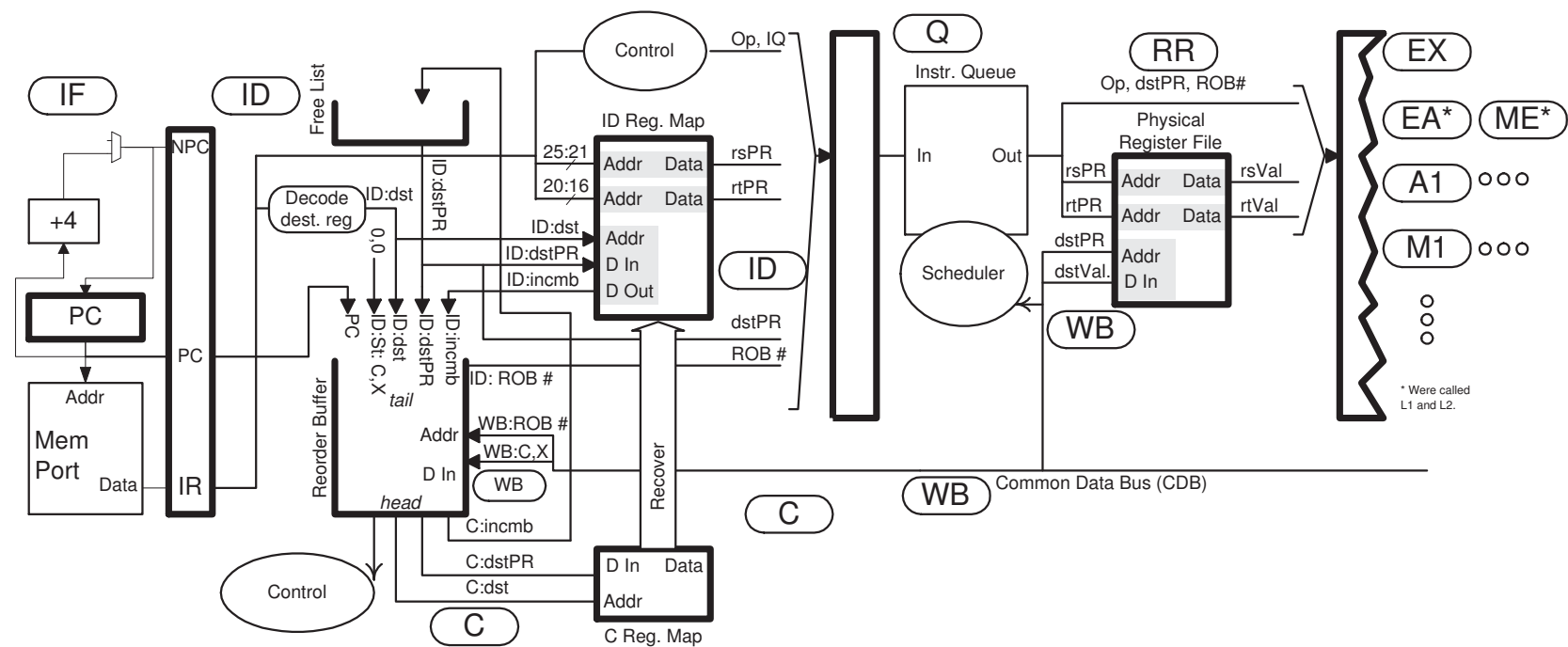
Increases instruction throughput of certain codes ...  
... such as those hitting the L2 cache.

A standard technique in general-purpose CPUs (desktop, laptop, server).

Even so, does not increase maximum practical superscalar width.

Details of dynamic scheduling not covered in this course.

Dynamically Scheduled Core



## Dynamically Scheduled Core Points

Needs more stages than statically scheduled cores.

Instruction scheduler consumes a significant amount of energy.

Energy wasted fetching down the wrong path of a predicted branch.



## Dynamically Scheduled Core Uses

Statically Scheduled : Dynamically Scheduled Core

Static: Relies on compiler to avoid stalls.

Dynamic: Handles insn with unpredictable latencies (loads).

## Good Code, Bad Code

Obviously Good Code (For the core, not the memory system.)

```
sum = 0;  
for ( int i=0; i<1000; i++ ) sum += b[i];
```

Easy to exploit ILP.

## Obviously Bad Code

```
for ( Node *node = start; node; node = node->next ) sum += node->data;
```

Can't overlap addition.

## More Bad Code

### Control Predictability

```
for ( int i=0; i<1000; i++ )
{
    switch ( op[i] ) {
        case ADD: a[i] = b[i] + c[i]; break;
        case DIV: a[i] = b[i] / c[i]; break;
    }
}
```

## Superscalar Microarchitecture Features

### *Branch Prediction*

The prediction of the direction (taken or not taken) and the target of a branch.

A standard technique in CPUs.

Prediction accuracy  $\approx 95\%$  for integer codes.

Prediction accuracy  $\approx 99.9\%$  for many scientific codes.

Branch prediction not covered in this course...

... because GPUs doesn't need it.

### *Vector Instructions:*

Instructions that operate on short vectors.

Many instruction sets have vector instructions ...

... often part of an instruction set *extension* ...

... such as *SSE* and *AVX* for Intel ...

... and *VIS* for Sun and *VFP* & *Advanced [tm] SIMD* for ARM.

### *Vector Register:*

A register holding multiple values.

Of course, vector instructions operate on vector registers.

## Vector Instruction Example — Favorable Case

Example is for a hypothetical instruction set.

Vector registers are **v0-v31**.

Each vector register holds four scalars.

Code **not** using vector instructions.

```
add.s f1, f2, f3
add.s f4, f5, f6
add.s f7, f8, f9
add.s f10, f11, f12
add.s f13, f14, f15
add.s f16, f17, f18
add.s f19, f20, f21
add.s f22, f23, f24
```

Code using vector registers.

```
# Register v2 holds equivalent of { f2, f5, f8, f11}.
add.vs v1, v2, v3    # Performs 4 adds, same work as first 4 insns above.
add.vs v4, v5, v6
```

## Vector Instruction Example — Unfavorable Case

Example is for a hypothetical instruction set.

Can't make full use of vector insn ...

... because can't find four of the same operation.

Code **not** using vector instructions.

```
add.s f1, f2, f3
add.s f4, f5, f6
mul.s f7, f8, f9
sub.s f10, f11, f12
```

Code using vector registers.

```
# v2 holds { f2, f5, X, X } (Two dummy values.)
add.vs v1, v2, v3    # Does 4 adds, but only two are useful.
mul.s v7, v8, v9    # Does 1 mul, but uses vector regs (as if they were scalar).
sub.s v10, v11, v12
```

:- ( Only one fewer instruction. Worth the trouble?

## Using Vector Instructions in Your Code

It's often **hard to find opportunities** to use vector instructions.

Compilers can do it ...

... but are often frustrated by unwitting programmers.

More coverage of vector instructions later in the semester ...

... including how **not to be** one of *those* programmers.

## Benefit of Vector Instructions

Lower cost (compared to scalar functional units.)



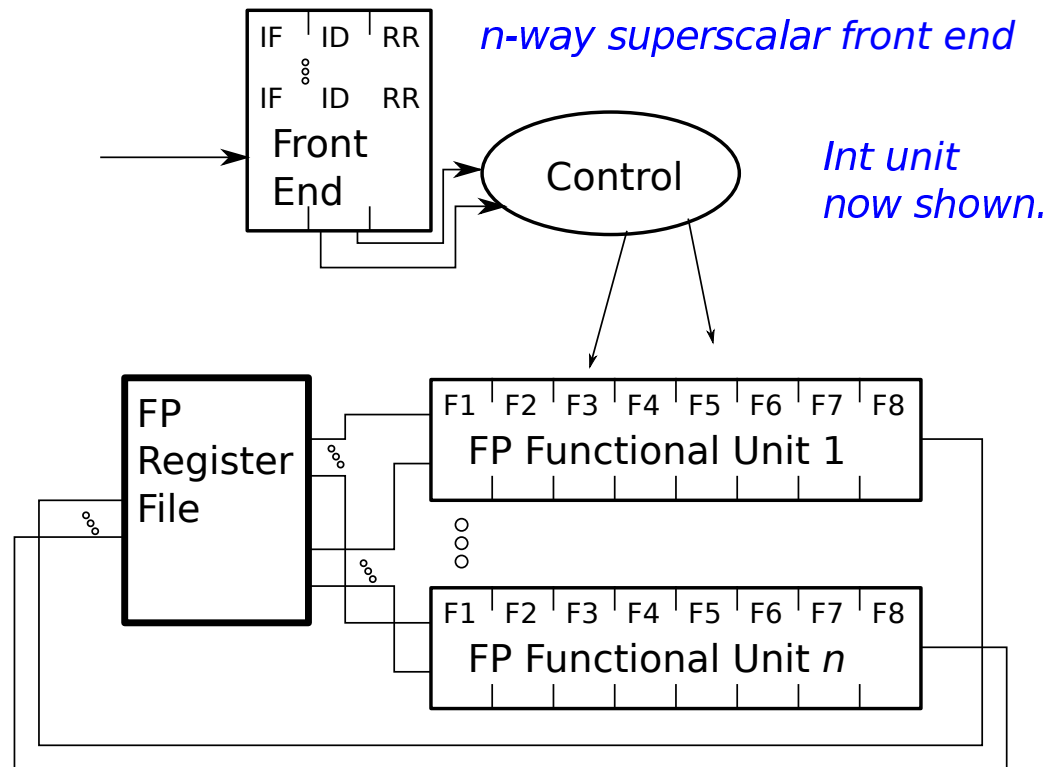
Goal:...

... want  $n$  FLOP/cyc...

... but don't need flexibility.

Consider this  $n$ -way superscalar core  $\rightarrow$

Can execute **any mix of  $n$  FP ops** per cycle...  
... if dependencies cooperate.

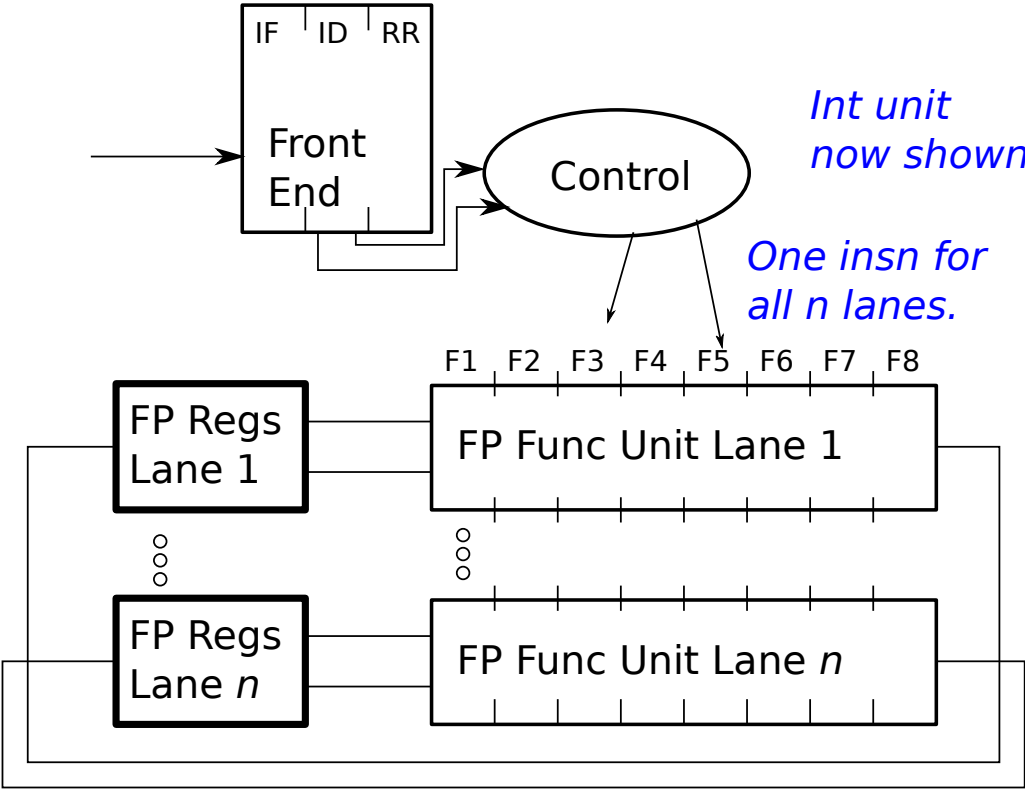


### Execution of Non-Vector Code on 4-Way Superscalar Core

Time / Cyc -->	0	1	2	3	4	5	6	7	8	9	10	11
add.s f1, f2, f3	IF	ID	RR	F1	F2	F3	F4	F5	F6	F7	F8	
add.s f4, f5, f6	IF	ID	RR	F1	F2	F3	F4	F5	F6	F7	F8	
add.s f7, f8, f9	IF	ID	RR	F1	F2	F3	F4	F5	F6	F7	F8	
add.s f10, f11, f12	IF	ID	RR	F1	F2	F3	F4	F5	F6	F7	F8	
add.s f13, f14, f15		IF	ID	RR	F1	F2	F3	F4	F5	F6	F7	F8
add.s f16, f17, f18		IF	ID	RR	F1	F2	F3	F4	F5	F6	F7	F8
add.s f19, f20, f21		IF	ID	RR	F1	F2	F3	F4	F5	F6	F7	F8
add.s f22, f23, f24		IF	ID	RR	F1	F2	F3	F4	F5	F6	F7	F8

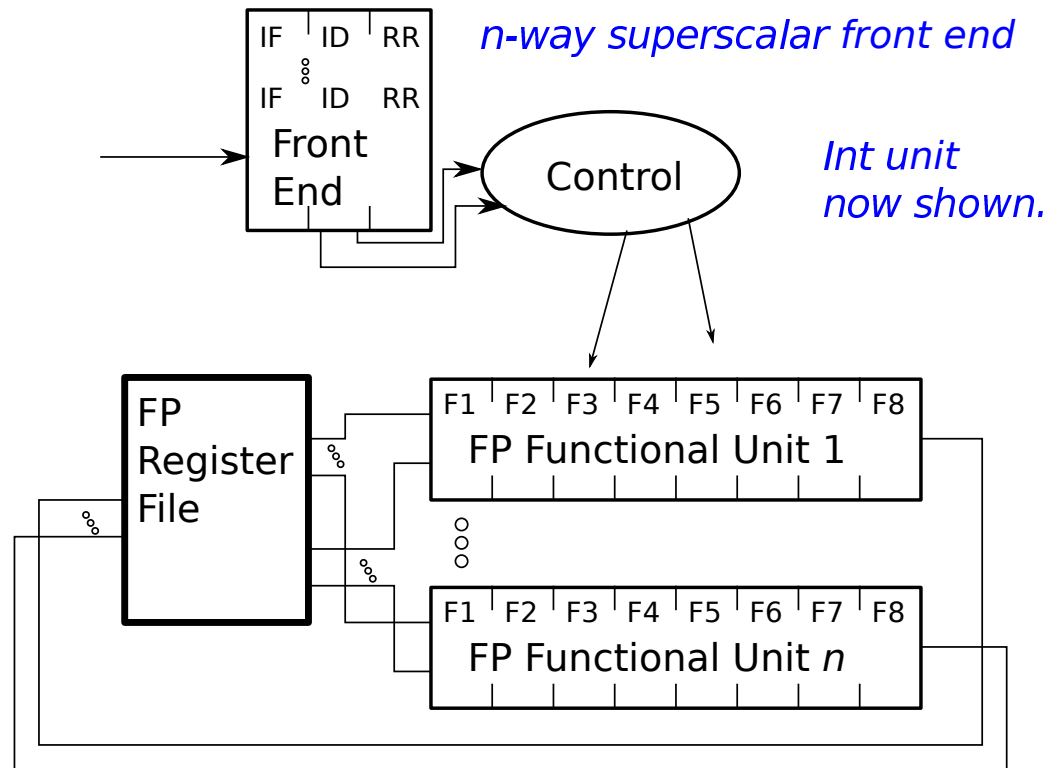
Vector Functional Unit:

A unit that performs the same operation on multiple sets of operands.



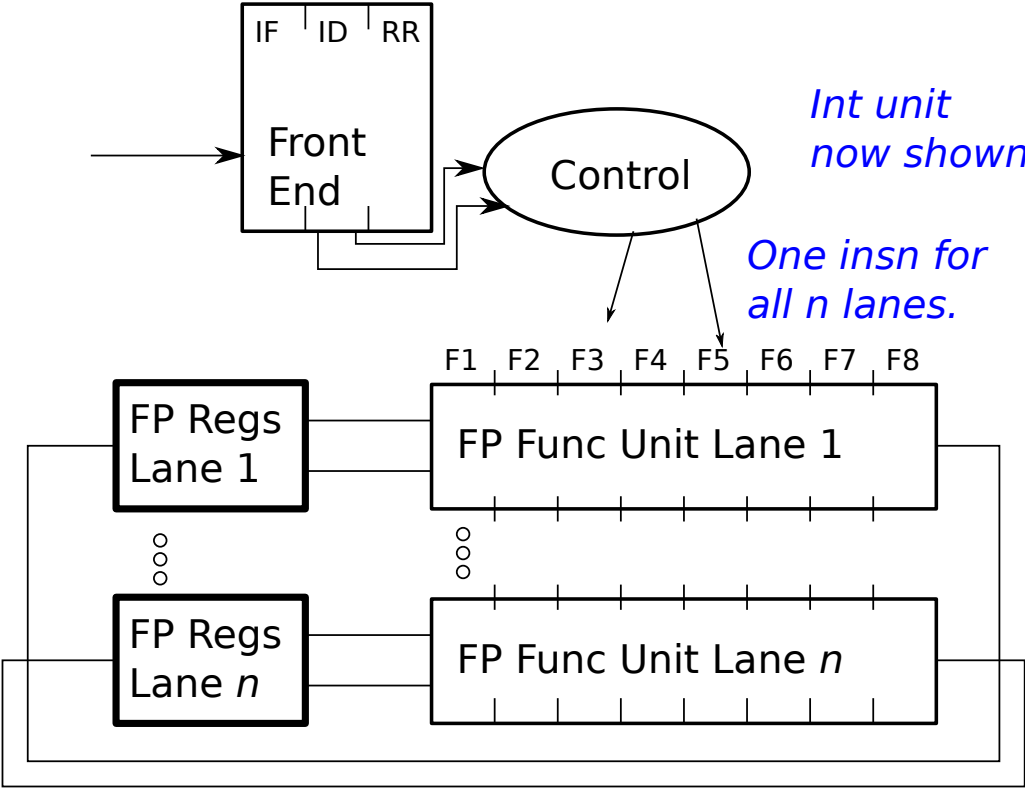
Execution of Code on a Scalar Core with a 4-Lane Vector Unit

Time / Cyc -->	0	1	2	3	4	5	6	7	8	9	10	11
add.vs v1, v2, v3	IF	ID	RR	F1	F2	F3	F4	F5	F6	F7	F8	
add.vs v4, v5, v6		IF	ID	RR	F1	F2	F3	F4	F5	F6	F7	F8



### Unsuitable Vector Code on 4-Way Superscalar

Time / Cyc -->	0	1	2	3	4	5	6	7	8	9	10	11
add.s f1, f2, f3	IF	ID	RR	F1	F2	F3	F4	F5	F6	F7	F8	
add.s f4, f5, f6	IF	ID	RR	F1	F2	F3	F4	F5	F6	F7	F8	
mul.s f7, f8, f9	IF	ID	RR	F1	F2	F3	F4	F5	F6	F7	F8	
sub.s f10, f11, f12	IF	ID	RR	F1	F2	F3	F4	F5	F6	F7	F8	



Unsuitable Vector Code on Scalar Core with a 4-Lane Vector Unit

Time / Cyc -->	0	1	2	3	4	5	6	7	8	9	10	11	12
add.vs v1, v2, v3	IF	ID	RR	F1	F2	F3	F4	F5	F6	F7	F8		
mul.s v7, v8, v9		IF	ID	RR	F1	F2	F3	F4	F5	F6	F7	F8	
sub.s v10, v11, v12			IF	ID	RR	F1	F2	F3	F4	F5	F6	F7	F8

## Vector Unit v. Superscalar

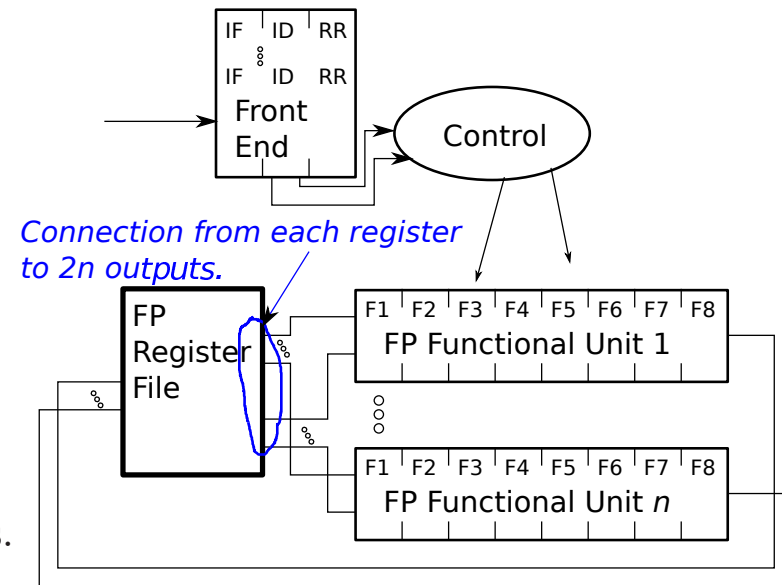
### Hypothetical Configurations to Compare

Both have a FP operation bandwidth of  $n$  per cycle.

Both have enough registers for  $R$  values.

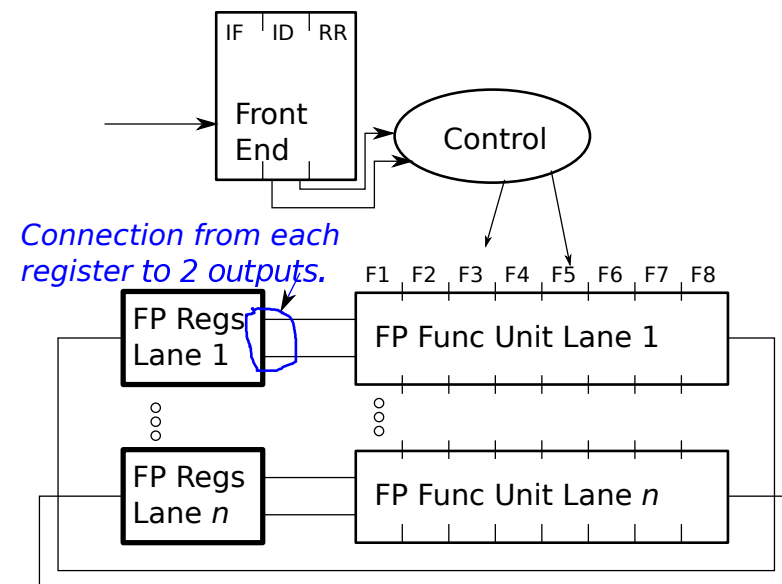
In superscalar there are  $R$  registers.

In vector system there are  $R/n$  regs each holding  $n$  values.



### Comparison

- Front end for superscalar costs  $n \times$  more.
- Superscalar bypass network:  $O(n^2)$ .
- Vector bypass network:  $O(n)$ .
- Superscalar reg fanout:  $R$  to  $2n$ .
- Vector reg fanout:  $R/n$  to 2.



## Vector Unit v. Superscalar

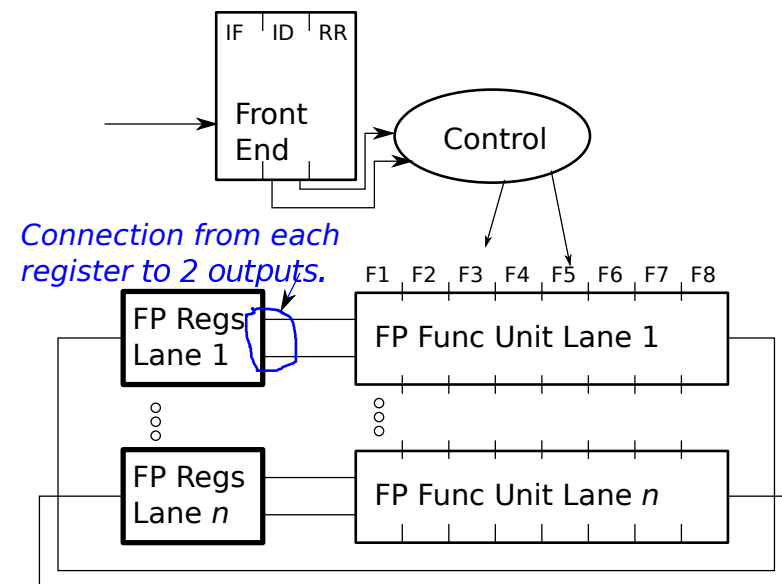
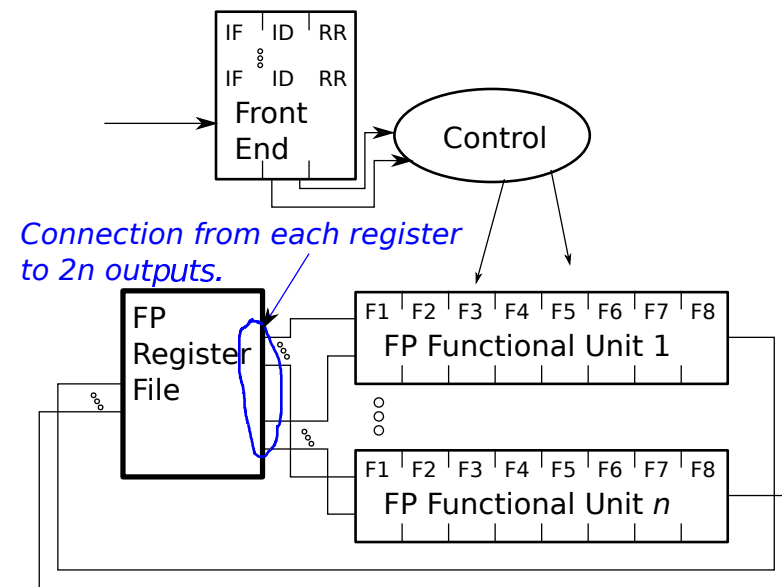
Hardware Limit on Width ( $n$ ).

Superscalar:  $O(n^2)$  cost.

Fanout impact on clock frequency...  
... maximum reasonable width  $\leq 8$ .

Vector:

Clock synchronization (reasons not covered in this course).



## Vector Units in Current Processors

Intel (i5, i7)

Four-Way Superscalar (based on *microops*, etc.).

SSE: Vector Registers (**xmm**):  $16 \times 128$  bits (4 SP, 2 DP).

AVX: Vector Registers (**ymm**):  $16 \times 256$  bits (8 SP, 4 DP).

*Process:*

A unit of execution managed by the OS, think of it as a running program.

A process has one address space.

A process can have multiple threads (which share the address space).

*Multiprocessing:*

A technique in which a core (or chip or node) can be shared by multiple processes, with an OS scheduler starting and stopping processes to achieve fairness, meet some priority goal, etc.

We **will not** consider multiple processes.

*Multithreading:*

A technique in which a single process can have multiple threads of execution, all sharing one address space.



*Context:*

The information associated with a thread or process, including the values of registers.

In this class *context* will refer to threads.

The context for a process is larger than the context for a thread.

*Context Switch:*

The process of changing from one thread to another.

## Types of Multithreading

### *Software Multithreading:*

A form of multithreading in which a context switch is performed by software.

Context switch performed by OS or by user code.

Context switch achieved by copying register values to and from *stack*.

Context switch can take hundreds of cycles:

CPU Context: 32 64-bit integer registers.

32 64-bit control registers.

32 64-bit floating-point registers.

Total amount of data to move:  $2 \times 96 \times 8 = 1536$  B...

... might require 192 instructions just for data copies.

### *Hardware Multithreading:*

A form of multithreading in which CPU core holds multiple contexts ...  
... and in which context switch very fast or not needed.

Core has multiple sets of registers, one for each context.

For CPUs, number of contexts is small, two to four.

Usually used with software multithreading.

## Reasons for Multiple Threads per Core

- Want simultaneous progress on multiple activities.
- Latency hiding.

### *Latency Hiding:*

Doing useful work while waiting for something to finish.

Term broadly applied:

*“Your call is important to us. Please stay on the line ...”*

Your attempt to report a problem with your Internet service is delayed ...  
... so you switch to physics homework.

## Latencies We'd Like to Hide

500,000.000  $\mu$ s   Internet Network Delay

10,000.000  $\mu$ s   Disk Access

0.100  $\mu$ s   Memory Access (L2 Cache Miss)

0.001  $\mu$ s   Instruction Operation Latency (un bypassed)

## Latency Hiding of IO Activity

Examples: Disk and network activity.

Easy, because OS already in control...  
... and latencies are long (multiple milliseconds).

## Latency Hiding of Instruction Events

Examples: Cache miss ( $\approx 100$  ns), insn-to-insn latency ( $\approx 10$  ns).

Times are too short for software multithreading.

No convenient way to tell OS when to switch.

Neither is a problem for hardware multithreading.

*Simultaneous Multithreading (SMT)*

A.k.a. *Hyperthreading* (Intel).

A multithreading system in which the context switch time is zero.

Multiple contexts, including a PC for each context.

Each cycle, hardware decides which context to use for fetch.

Fetches instruction proceeds down pipeline ...

... next to insns from other contexts.

*Consider:*

# Single Thread:

T0: 0x1000    add r1, r2, r3    IF ID EX ME WB

T0: 0x1004    sub r4, r5, r6        IF ID EX ME WB

# Simultaneous Multithreaded

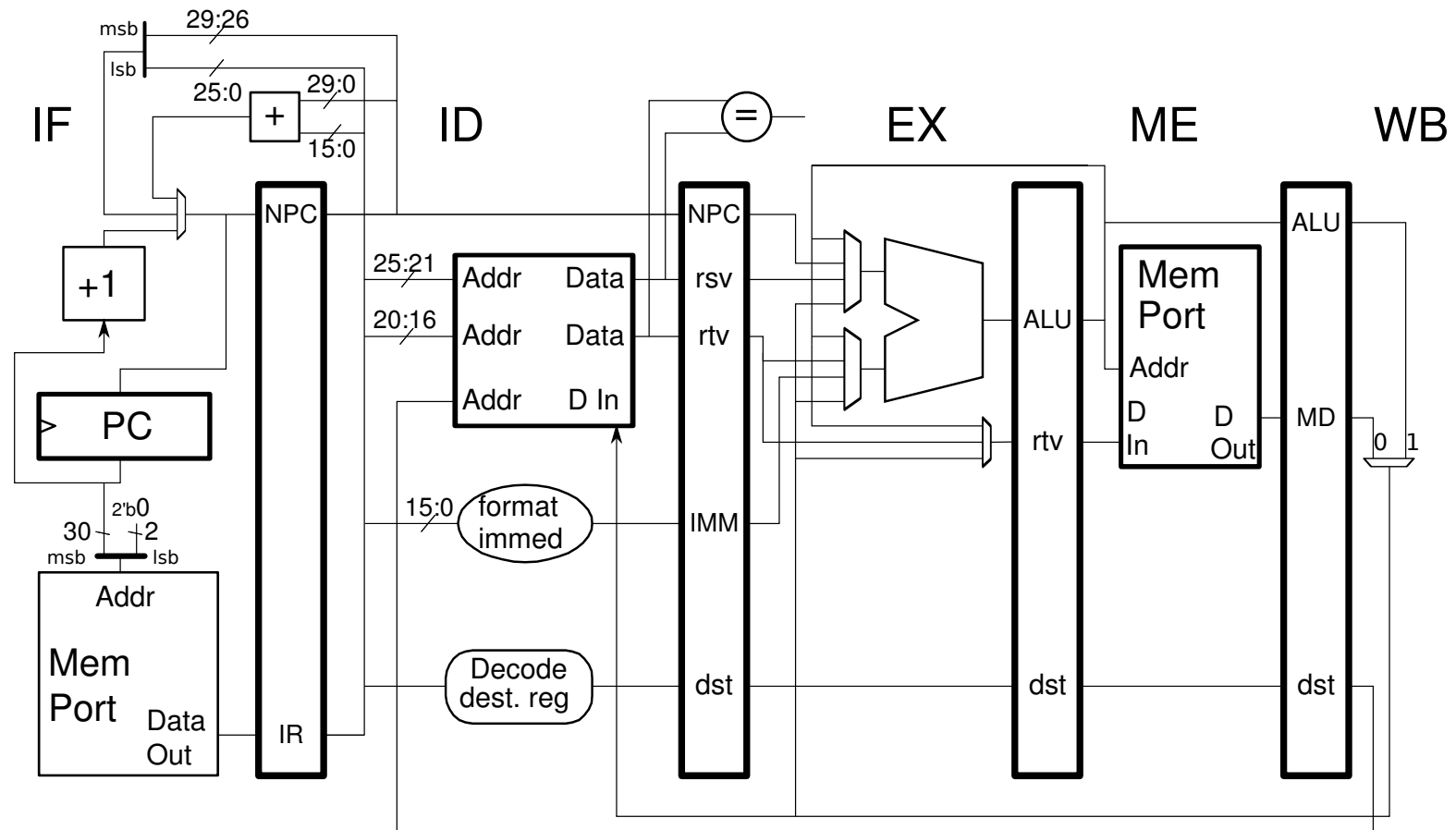
T0: 0x1000    add r1, r2, r3    IF ID EX ME WB

T1: 0x2000    sub r1, r2, r3        IF ID EX ME WB

## SMT Hardware

Easy to do in a dynamically scheduled system.

Consider the following statically scheduled system:





## SMT Hardware

Changes for an  $n$ -way SMT:

Replace PC with  $n$  PCs.

Increase register field by  $\lceil \log_2 n \rceil$  bits.

Limit squash and other insn events to correct thread.

## Thread Selection

- Round Robin
- Fewer in flight

## Simultaneous Multithreading and CPUs

Number of contexts in current CPUs 2-4.

Can hide occasional latencies.

Often benefit of hiding latency ...

... less than problem of multiple threads occupying cache space ...

... that would otherwise be used by just one.

## Vector Core v. Superscalar Core

(We'll get back to multithreaded machines right after this.)

Because a superscalar core is more expensive we need to justify its use.

Vector core benefit: less expensive.

Superscalar core benefit: can run non-vectorizable code.

These tradeoffs are clear ...

... because few would deny that **non-vectorizable code** is common.

## SMT v. Single Thread

Because an SMT is more expensive we need to justify its cost.

Feature of SMT: Can hide latency.

Tough question to answer: is this the best way to hide latency?

Consider

If switching between two threads would hide latency ...

... then maybe the same latency could be hidden by combining threads.

Combining threads is effective if there are enough registers.

Weak Example for SMT

Consider two scalar cores, one with 5-way SMT.

```
# Single-Thread      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
T0: mul.s f0, f2, f4  IF ID RR F1 F2 F3 F4
T0: add.s f6, f0, f6   IF ID -----> RR F1 F2 F3 F4
T0: mul.s f10, f12, f14 IF -----> ID RR F1 F2 F3 F4
T0: add.s f16, f10, f16                IF ID -----> RR F1 F2 F3 F4
..
T0: add.s f46, f40, f46                                Cyc 25-> IF

# Five Thread SMT    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
T0: mul.s f0, f2, f4  IF ID RR F1 F2 F3 F4
T1: mul.s f0, f2, f4   IF ID RR F1 F2 F3 F4
T2: mul.s f0, f2, f4   IF ID RR F1 F2 F3 F4
T3: mul.s f0, f2, f4   IF ID RR F1 F2 F3 F4
T4: mul.s f0, f2, f4   IF ID RR F1 F2 F3 F4
T0: add.s f6, f0, f6   IF ID RR F1 F2 F3 F4
T1: add.s f6, f0, f6   IF ID RR F1 F2 F3 F4
T2: add.s f6, f0, f6   IF ID RR F1 F2 F3 F4
T3: add.s f6, f0, f6   IF ID RR F1 F2 F3 F4
T4: add.s f6, f0, f6   IF ID RR F1 F2 F3 F4
#                      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
```

Yay!!! SMT Wins!!!

Weak Example for SMT

Reschedule (rearrange) Single-Thread Code

```
# Single-Thread      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
T0: mul.s f0, f2, f4  IF ID RR F1 F2 F3 F4
T0: mul.s f10, f12, f14 IF ID RR F1 F2 F3 F4
T0: mul.s f20, f22, f24      IF ID RR F1 F2 F3 F4
T0: mul.s f30, f32, f34      IF ID RR F1 F2 F3 F4
T0: mul.s f40, f42, f44      IF ID RR F1 F2 F3 F4
T0: add.s f6, f0, f6          IF ID RR F1 F2 F3 F4
T0: add.s f16, f10, f16      IF ID RR F1 F2 F3 F4
..
T0: add.s f46, f40, f46      IF ID RR F1 F2 F3 F4

# Five Thread SMT    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
T0: mul.s f0, f2, f4  IF ID RR F1 F2 F3 F4
T1: mul.s f0, f2, f4      IF ID RR F1 F2 F3 F4
T2: mul.s f0, f2, f4      IF ID RR F1 F2 F3 F4
T3: mul.s f0, f2, f4      IF ID RR F1 F2 F3 F4
T4: mul.s f0, f2, f4      IF ID RR F1 F2 F3 F4
T0: add.s f6, f0, f6      IF ID RR F1 F2 F3 F4
T1: add.s f6, f0, f6      IF ID RR F1 F2 F3 F4
T2: add.s f6, f0, f6      IF ID RR F1 F2 F3 F4
T3: add.s f6, f0, f6      IF ID RR F1 F2 F3 F4
T4: add.s f6, f0, f6      IF ID RR F1 F2 F3 F4
#                          0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
```

Now it's a tie, assuming that there are enough registers for scheduling.

Good Example for SMT

A load suffers a cache miss ...  
... compiler can't schedule around that ...  
... because misses are rarely predictable ...  
... and don't occur every occurrence.

# Cycle	Single Thread	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T0: lw f1, [r2]		IF	ID	RR	EX	ME	WB										
T0: lw f11, [r2+4]			IF	ID	RR	EX	ME										
T0: add.s f0, f1, f0				IF	ID	->	RR										
T0: add.s f10, f11, f10					IF	->	ID										

# Cycle	Multi Thread	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T0: lw f1, [r2]		IF	ID	RR	EX	ME	WB										
T1: lw f11, [r2+4]			IF	ID	RR	EX	ME										
T0: add.s f0, f1, f0				IF	ID	->	RR	F1	F2	F3	F4						
T0: sub.s f2, f3, f4					IF	->	ID	RR	F1	F2	F3	F4					

Notice that T0 can make progress while T1 waits.



### *Single-Instruction Multiple Thread (SIMT):*

A technique in which threads are managed in groups (called warps) to simplify hardware, all threads in a warp execute the same instruction (or are masked out) .

SIMT coined by NVIDIA to describe organization of their GPUs.

Hardware similar to vector unit hardware ...

... in the sense that ...

... for the fetch of one instruction ...

... multiple operations are performed.

Software model closer to multicore ...

... in the sense that ...

... programmer works with ordinary scalar registers ...

... and does not need to think about vector registers.

## SIMT Characteristics

Core holds many threads (thread contexts).

Threads are organized into groups called *warps*.

As in an ordinary multithreaded system ...  
... each thread has its own PC.

Instruction fetch is performed for an entire warp ...  
... using the PC of one of the threads in the warp.

This works well when ...  
... all of the threads in a warp have the same PC.

If threads have different PC values process must be repeated.

## SIMT Thread Dispatch

*Dispatch:*

Sending a thread (in this case) to an execution unit.

In other core organizations dispatch was one cycle ...  
... but for SIMT can be multiple cycles.

## Choice of Type of Core

Look at these questions:

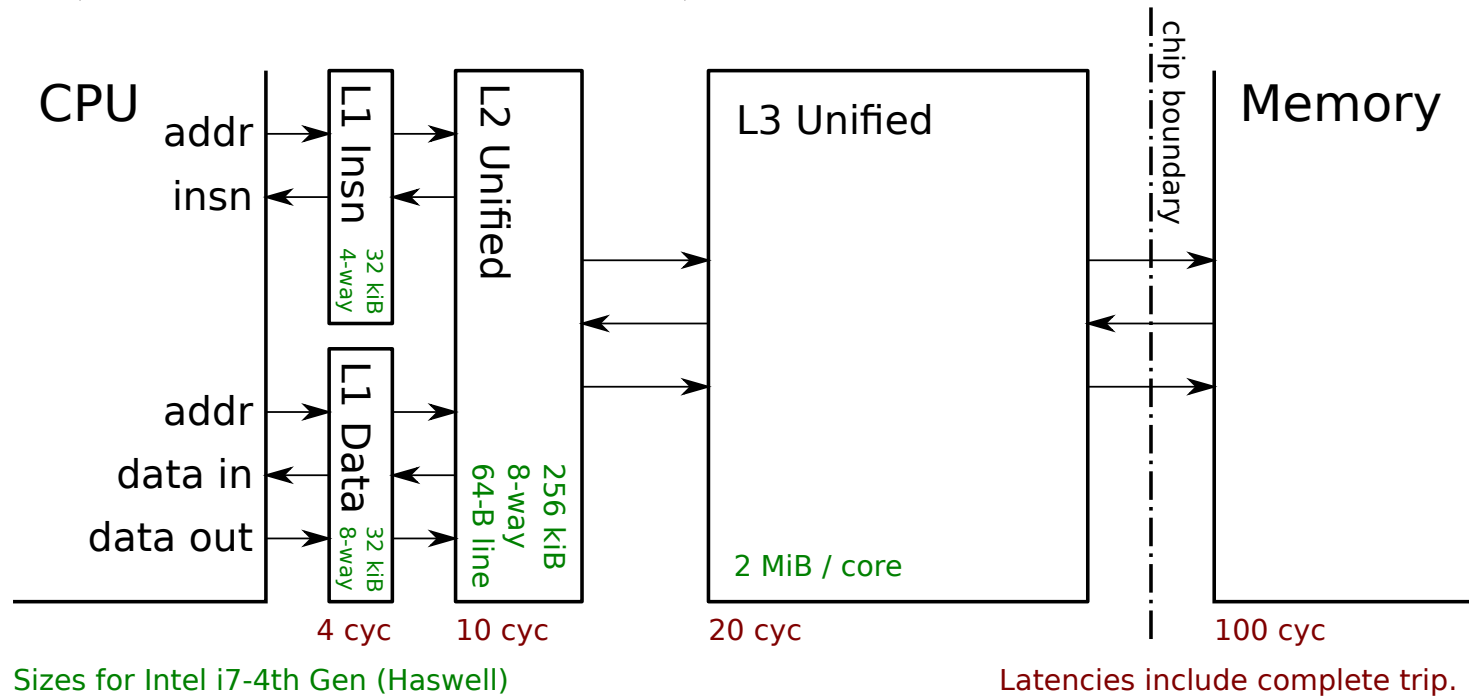
Type of parallelism?

Number of contextes?

Amount of cache?

## Typical Memory Hierarchy

- L1 (Level 1) Cache
- L2 (Level 2) Cache
- DRAM (Main Memory, Physical Memory)

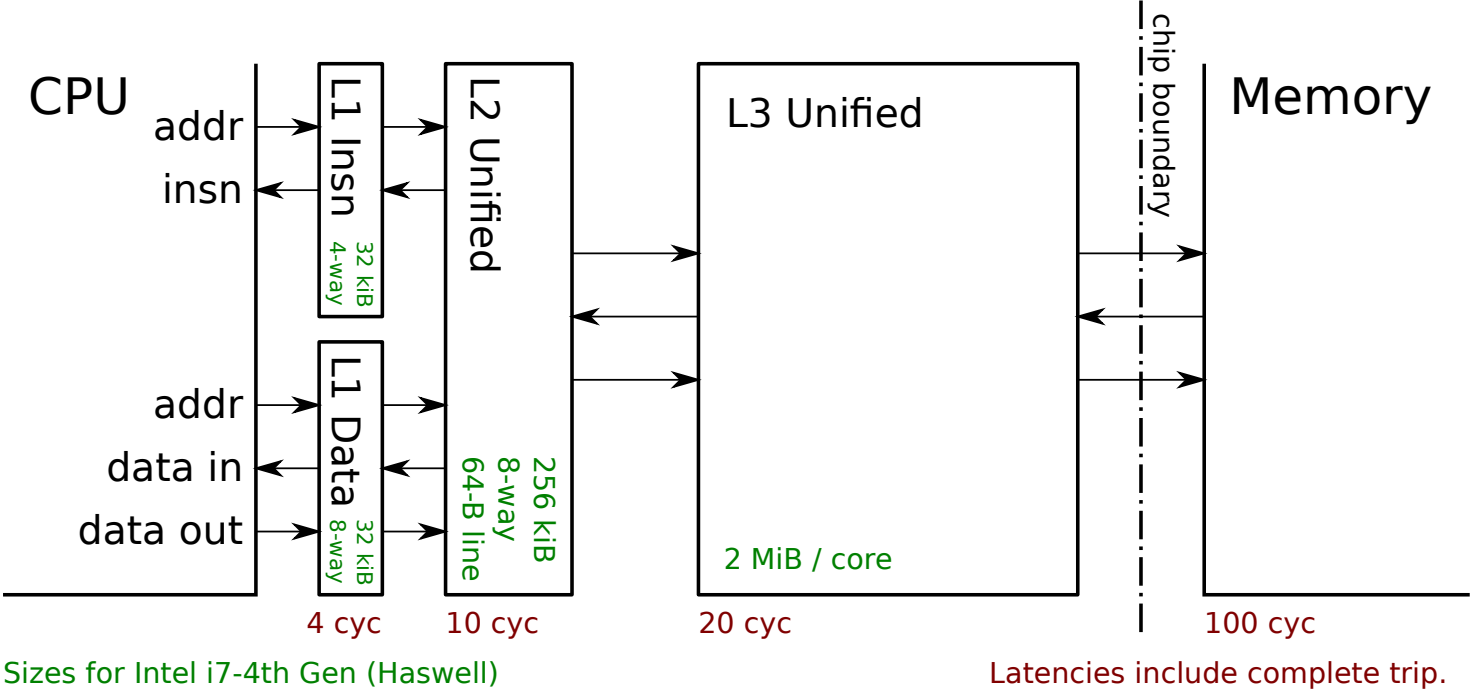


L1 Cache

First place checked.

Typical latency 1-2 cycles.

Each core has its own L1.



## L2 Cache

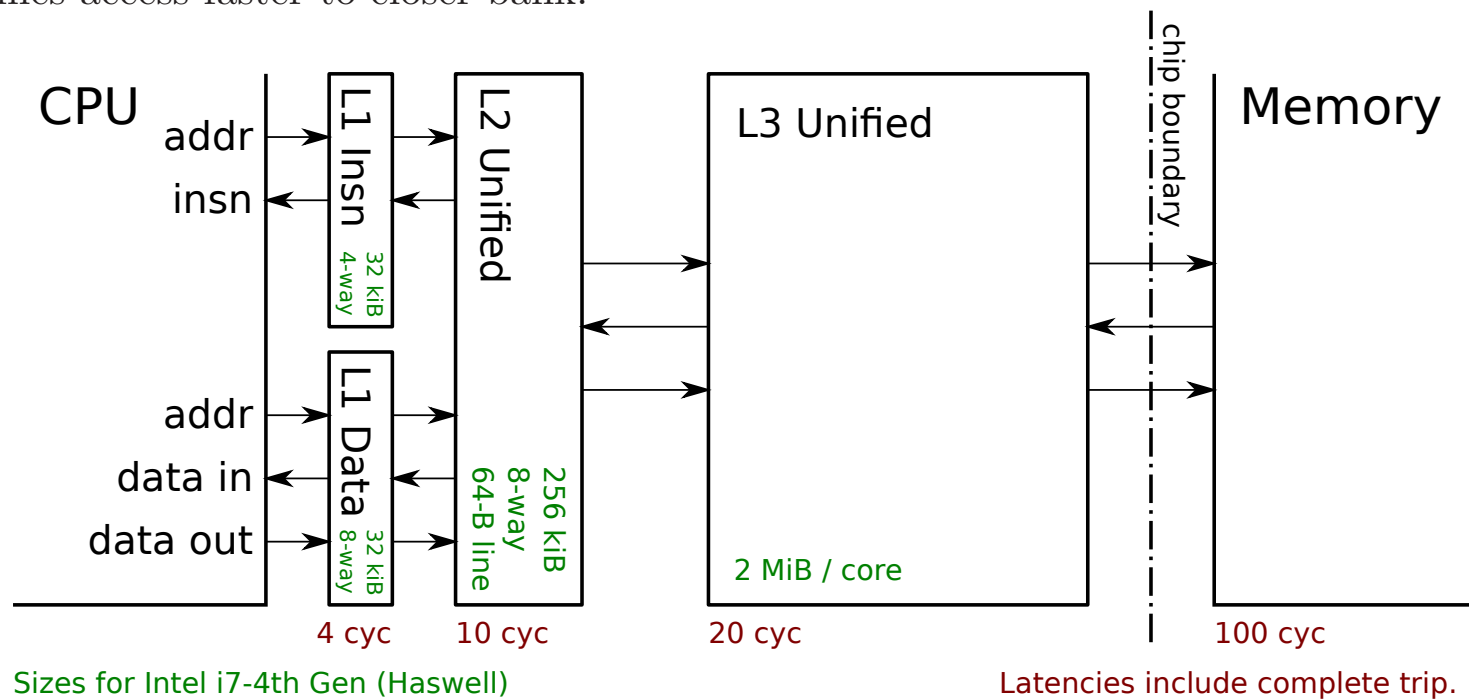
Checked on an L1 miss.

Typical latency 10-20 cycles.

Usually all cores on chip share L2.

L2 may be divided into *banks*.

Sometimes access faster to closer bank.



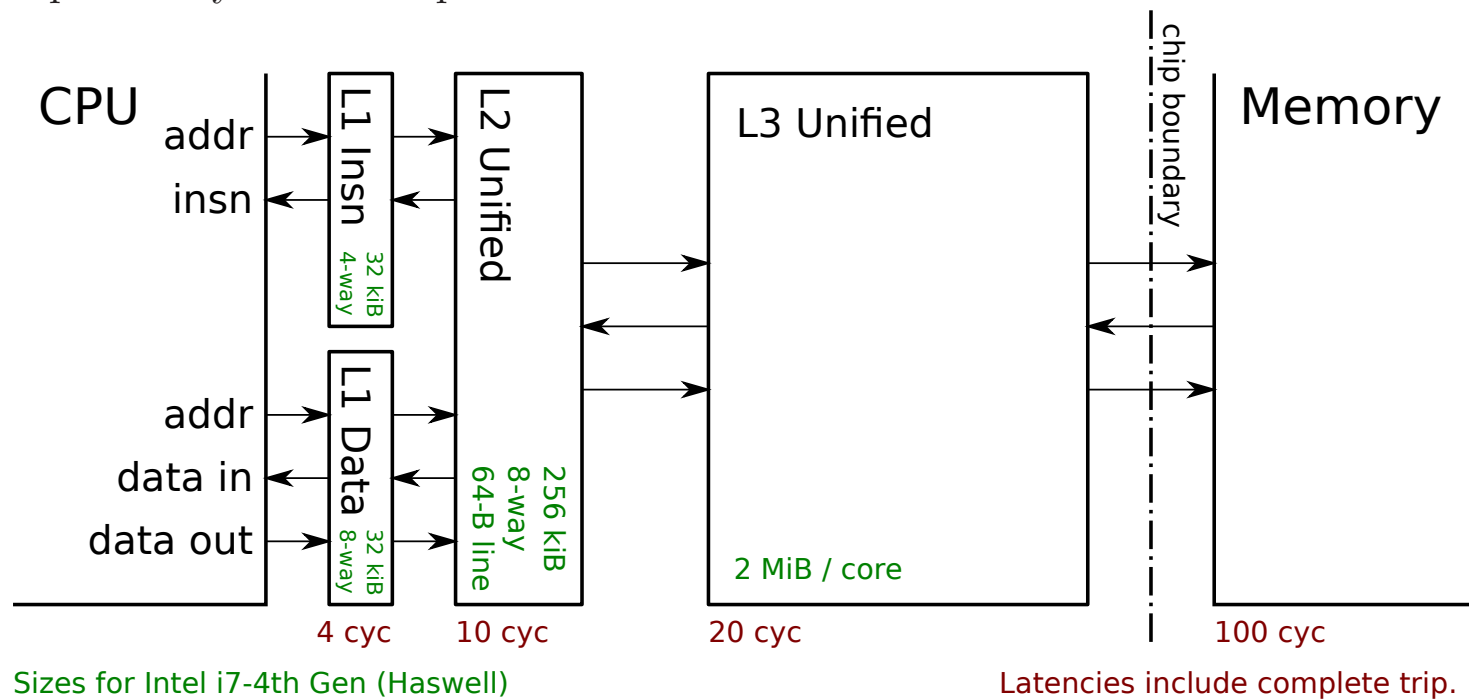
## DRAM

Holds the "original" value of address.

Typical latency 100-500 cycles.

Located off chip.

An on-chip memory controller provides access to DRAM.





## Typical Multicore Chip Organization

Each core has its own L1 cache.

L2 cache divided into banks.

Memory controllers connect to off-chip memory.

Communication between cores, L2, and MC via an *interconnect*.

## Interconnect Types

### *Crossbar*

Interconnect with a switch for each input/output pair.

Can always connect a free input to a free output.

Latency is  $O(1)$ .

Cost  $O(n^2)$ .

### *Ring*

Connections form a loop visiting all ports.

Delay is  $O(n)$ .

Cost is  $O(n)$ .

*Bus*

Single shared medium connecting all ports.

Only one pair can communicate at a time.

Latency is  $O(1)$ .

Cost is  $O(1)$ .

## Common Organizations

Heavy Multicore (i7, etc)

Crossbar interconnect.

Manycore (Xeon Phi, Sun Niagara)

Ring interconnect.