Beyond Pipelining

Virendra Singh
Associate Professor
Computer Architecture and Dependable Systems Lab
Department of Electrical Engineering
Indian Institute of Technology Bombay
http://www.ee.iitb.ac.in/~viren/
E-mail: viren@ee.iitb.ac.in

EE-739: Processor Design

Lecture 2 (15 Jan 2015)

CADSL

Very Large Instruction Word (VLIW) Architecture

VLIW was invented

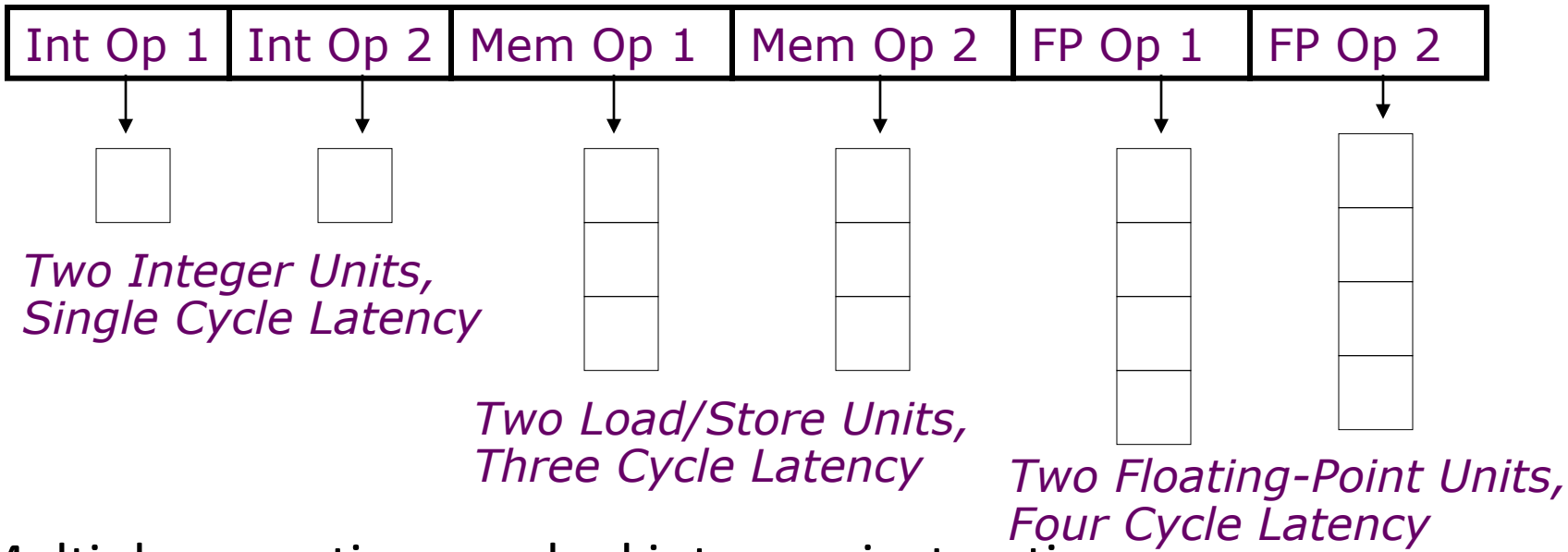
The idea of VLIW has been considered the work on trace scheduling, a method of compiling programs written in conventional languages for wide-word machines, done by Josh Fisher in 1979 at Yale laid down the foundation for VLIW technology. Now John Fisher leads HP's VLIW compiler project.

VLIW Pioneer: HP Senior Fellow Josh Fisher beside his MultiFlow Trace VLIW machine, on display at Computer History Museum.

Why VLIW ?

- To overcome the difficulty of finding parallelism in machine-level object code.
- In a VLIW processor, multiple instructions are packed together and issued in parallel to an equal number of execution units.
- The compiler (not the processor) checks that there are only independent instructions executed in parallel.

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
- Guarantees intra-instruction parallelism
- Schedules to avoid data hazards (no interlocks)
  - Typically separates operations with explicit NOPs

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

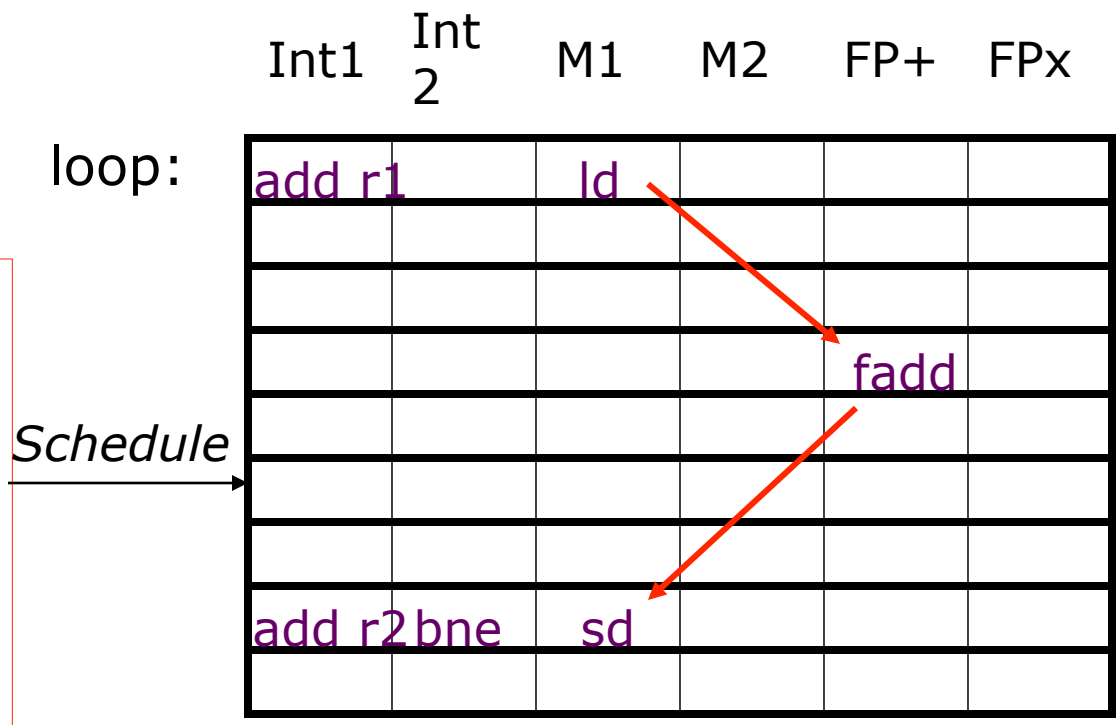
Loop Execution

```
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 →

	Int1	Int 2	M1	M2	FP+	FPx
loop:			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			

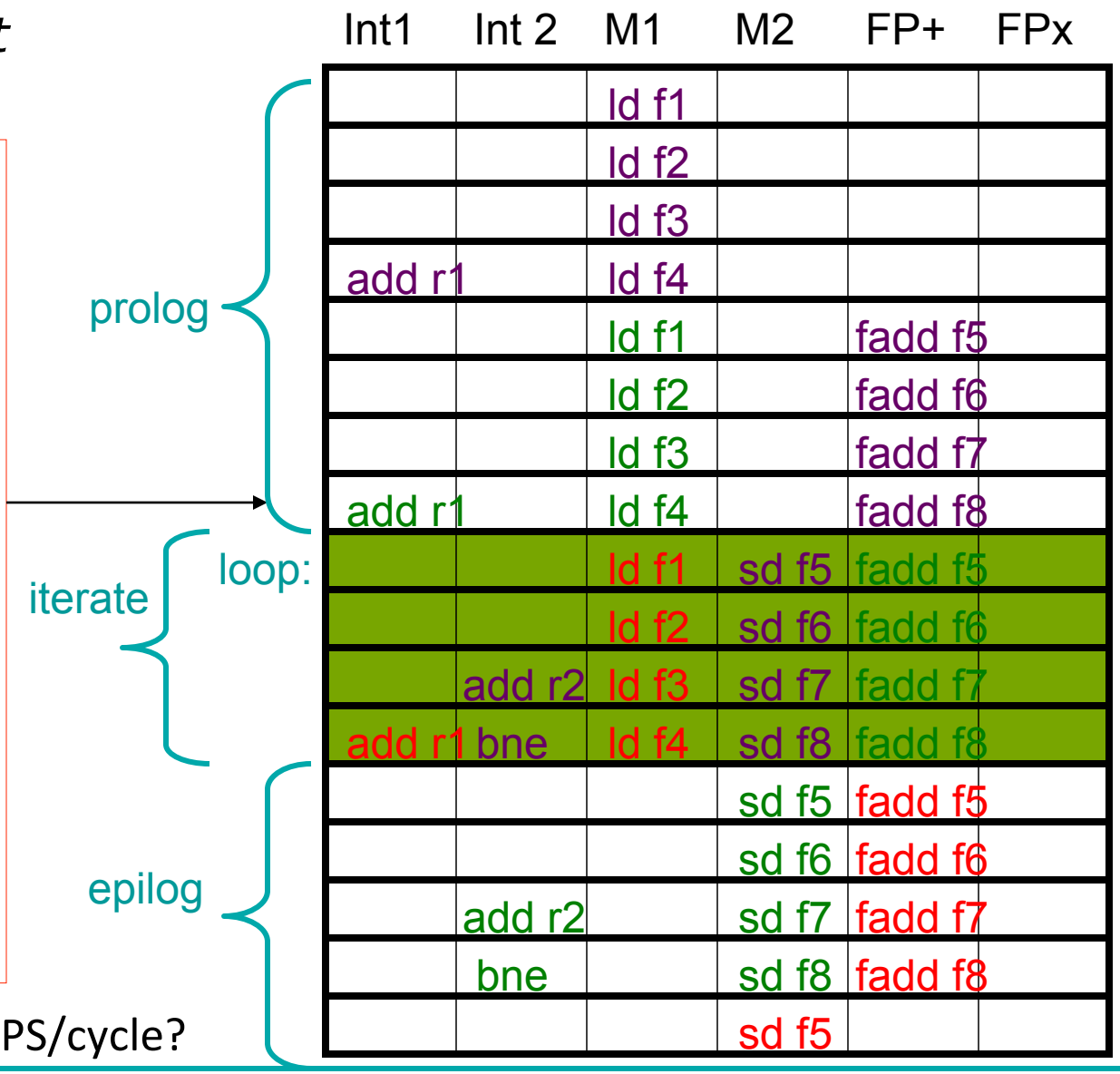
How many FLOPS/cycle? 4 fadds / 11 cycles = 0.36

Software Pipelining

Unroll 4 ways first

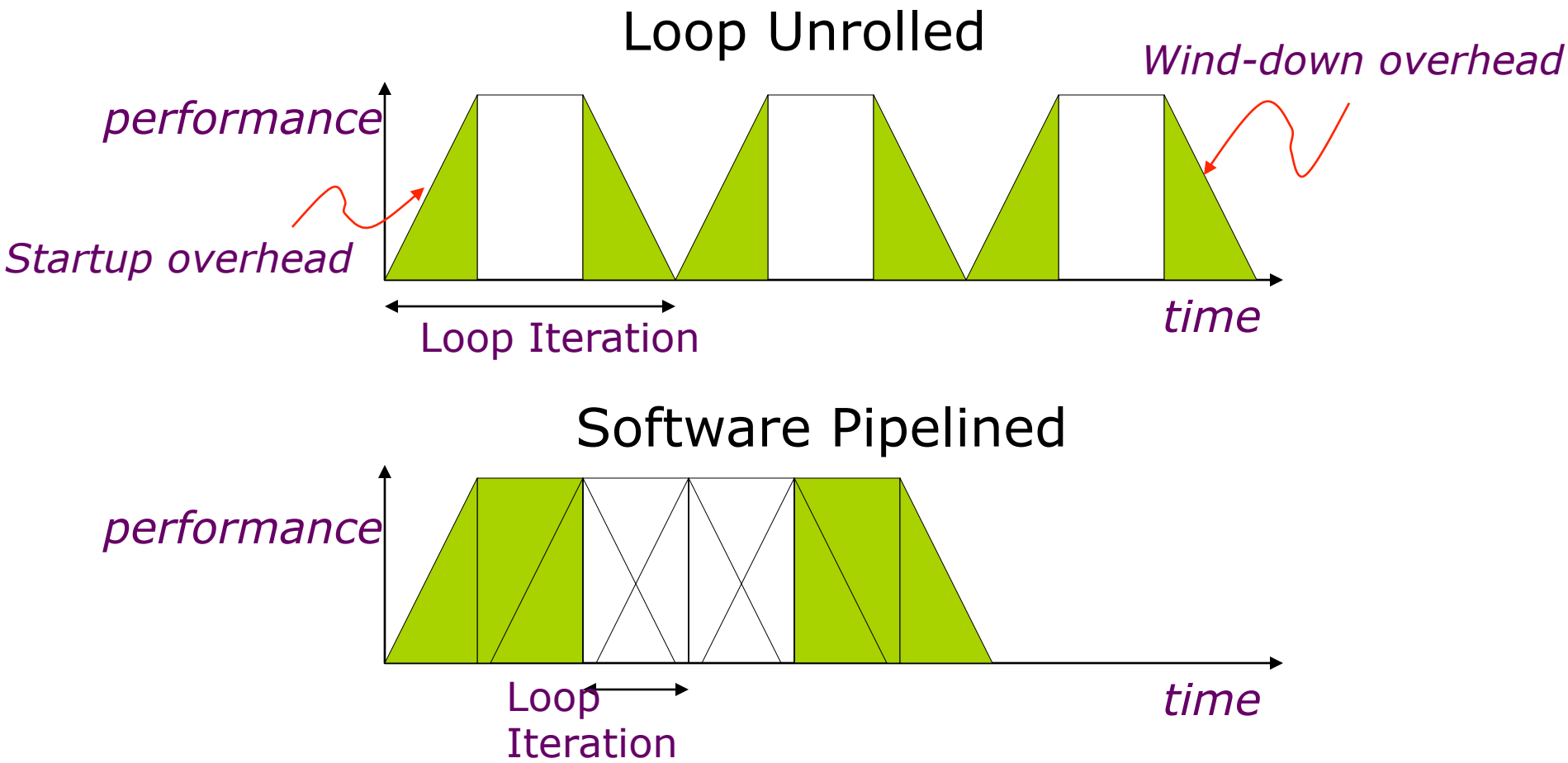
```
loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      add r2, 32
      sd f8, -8(r2)
      bne r1, r3,
loop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

How many FLOPS/cycle?



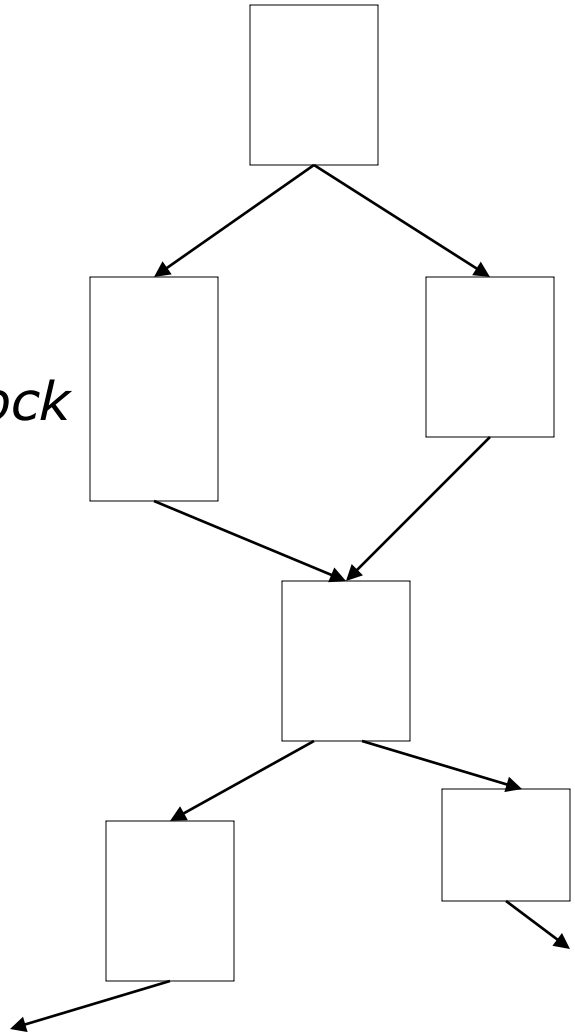
4 fadds / 4 cycles = 1

Software Pipelining vs. Loop Unrolling



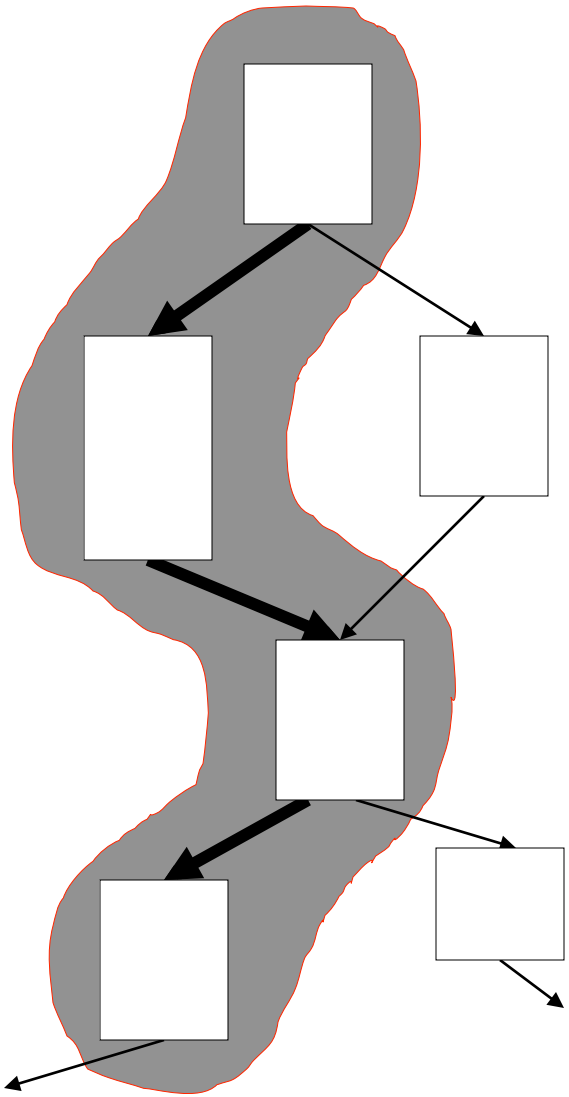
Software pipelining pays startup/wind-down costs only once per loop, not once per iteration

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 *[Fisher,Ellis]*

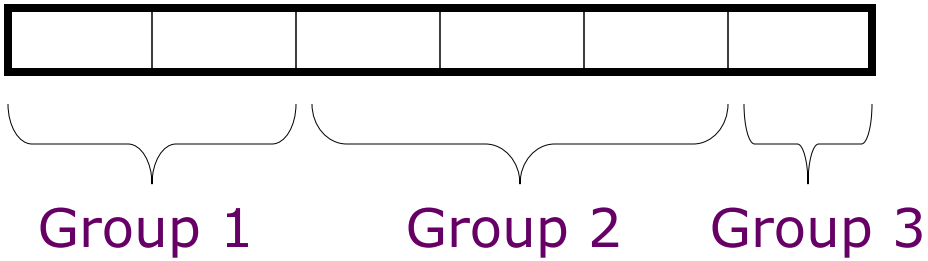


- Pick string of basic blocks, a *trace*, that represents most frequent branch path
- Use profiling feedback or compiler heuristics to find common branch paths
- Schedule whole "trace" at once
- Add fixup code to cope with branches jumping out of trace

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

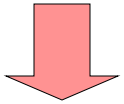


- Schemes to reduce effect of unused fields
  - Compressed format in memory, expand on I-cache refill
    - used in Multiflow Trace
    - introduces instruction addressing challenge
  - Mark parallel groups
    - used in TMS320C6x DSPs, Intel IA-64
  - Provide a single-op VLIW instruction
    - Cydra-5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

Rotating Register Files

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

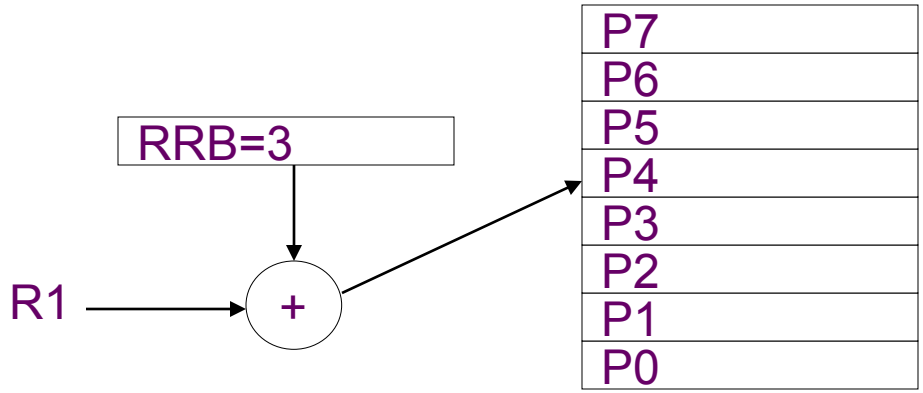
ld r1, ()		
add r2, r1, #1	ld r1, ()	
st r2, ()	add r2, r1, #1	ld r1, ()
	st r2, ()	add r2, r1, #1
		st r2, ()



Prolog	ld r1, ()		
	ld r1, ()	add r2, r1, #1	
Loop	ld r1, ()	add r2, r1, #1	st r2, ()
Epilog		add r2, r1, #1	st r2, ()
			st r2, ()

Solution: Allocate new set of registers for each loop iteration

Rotating Register File



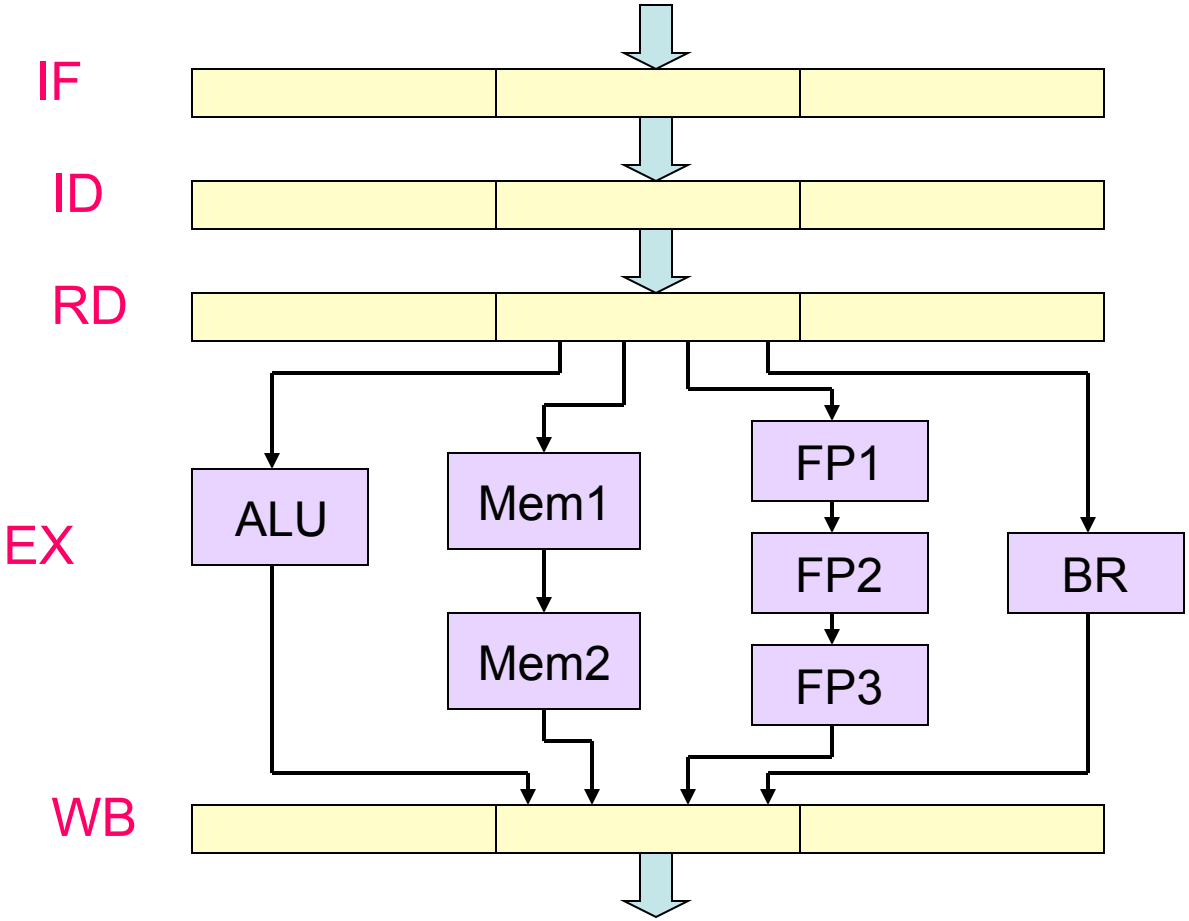
Rotating Register Base (RRB) register points to base of current register set. Value added on to logical register specifier to give physical register number. Usually, split into rotating and non-rotating registers.

Prolog	ld r1, ()			dec RRB
	ld r1, ()	add r3, r2, #1		dec RRB
Loop	ld r1, ()	add r3, r2, #1	st r4, ()	bloop
		add r2, r1, #1	st r4, ()	dec RRB
Epilog			st r4, ()	dec RRB

Loop closing branch decrements RRB

Superscalar Design

Superscalar Pipelines (Diversified)



Superscalar Pipelines (Diversified)

Diversified Pipelines

- Each pipeline can be customized for particular instruction type
- Each instruction type incurs only necessary latency
- Certainly less expensive than identical copies
- If all inter-instruction dependencies are resolved then there is no stall after instruction issue

Require special consideration

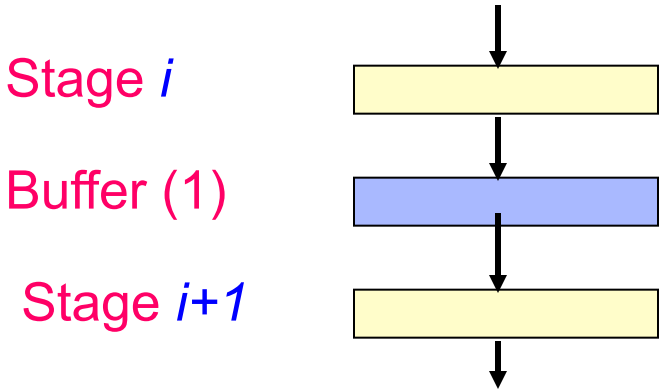
- Number and Mix of functional units

Superscalar Pipelines (Dynamic Pipelines)

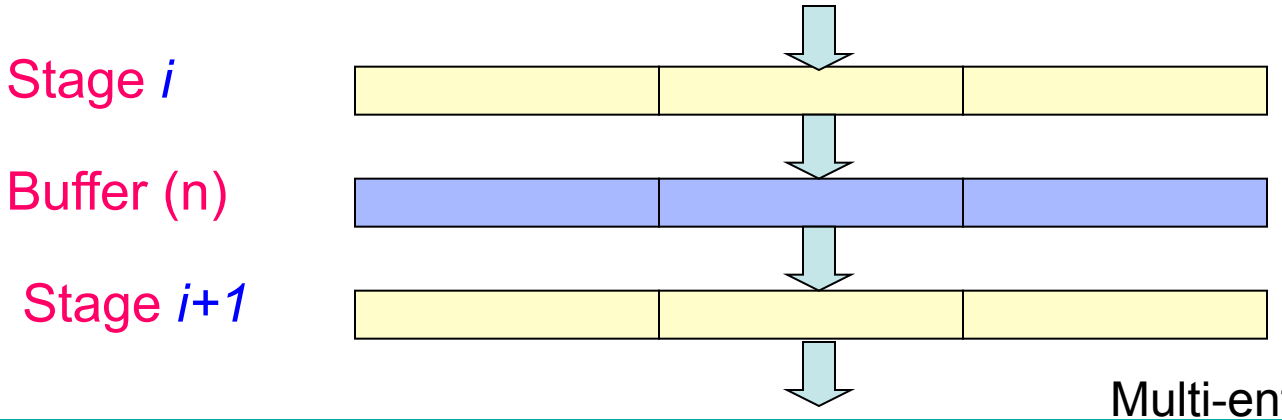
Dynamic Pipelines

- Buffers are needed
  - Multi-entry buffers
  - Every entry is hardwired to one read port and one write port
  - Complex multi-entry buffers
  - Minimize stalls

Superscalar Pipelines (Interpipeline-Stage Buffers)

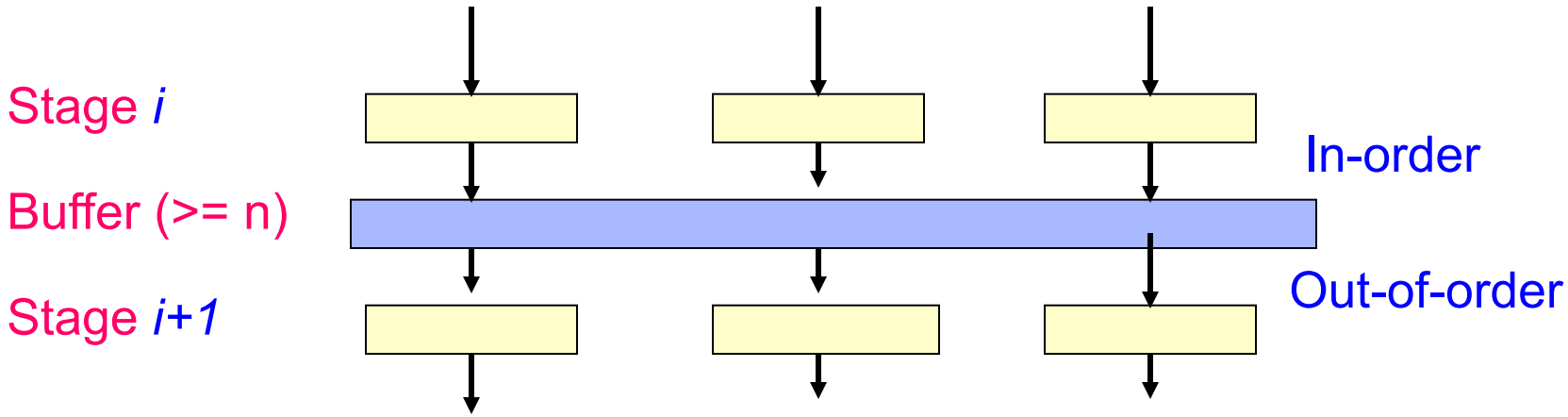


Single entry



Multi-entry Buffer

Superscalar Pipelines (Interpipeline-Stage Buffers)



- Trailing instructions are allowed to bypass stalled instruction
- Out-of-order execution

Superscalar Architecture

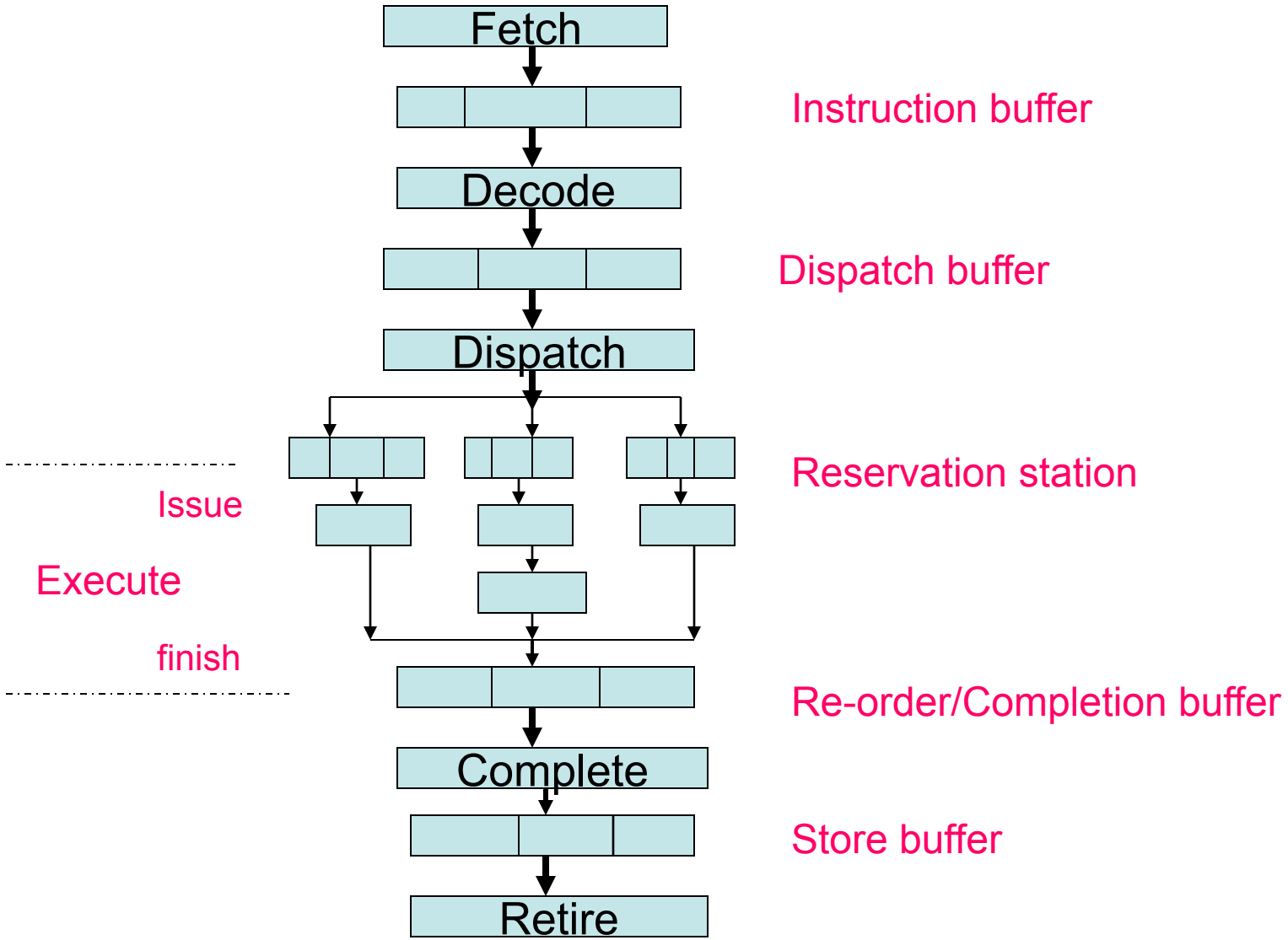
- Wide pipeline
- Superscalar architecture is natural descendant of pipelined scalar RISC
- Superscalar techniques largely concern the processor organization, independent of the ISA and the other architectural features
- Thus, possibility to develop a processor code compatible with an existing architecture

Superscalar Architecture

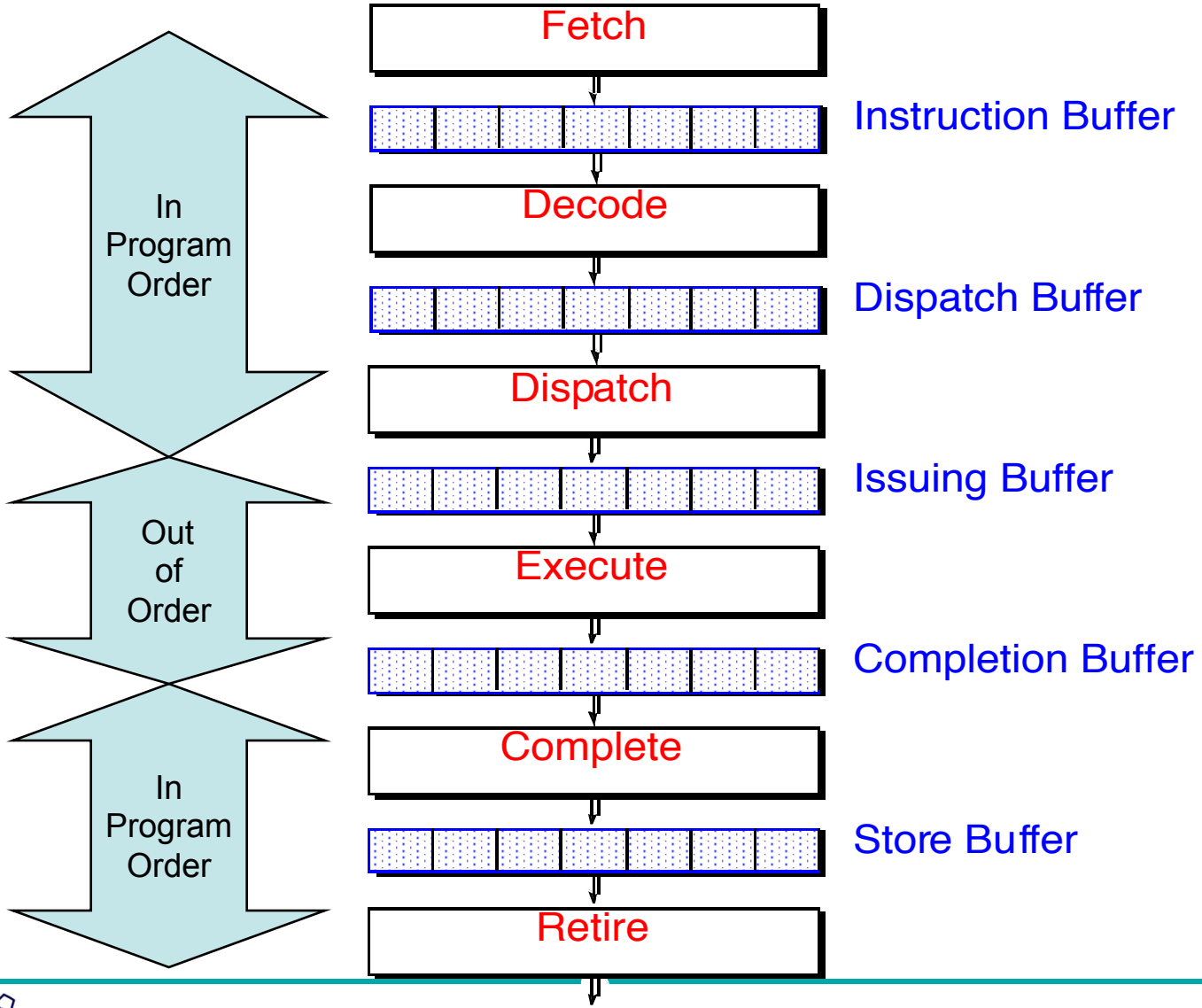
- Instruction issue and machine parallelism
  - ILP is not necessarily exploited by widening the pipelines and adding more resources
  - Processor policies towards fetching decoding, and executing instruction have significant effect on its ability to discover instructions which can be executed concurrently
  - Instruction issue is refer to the process of initiating instruction execution
  - Instruction issue policy limits or enhances performance because it determines the processor's look ahead capability

Highway

Super-scalar Architecture

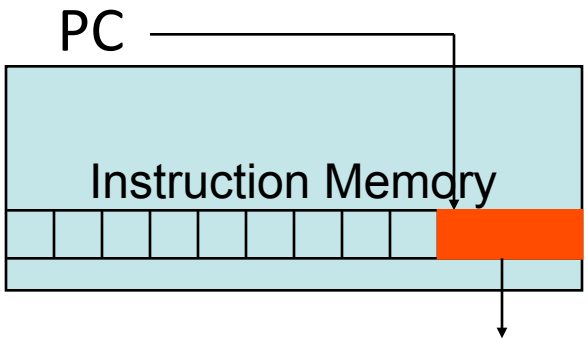


Superscalar Pipeline Stages



Instruction Flow

- Objective: Fetch multiple instructions per cycle
- Challenges:
  - Branches: control dependences
  - Branch target misalignment
  - Instruction cache misses



3 instructions fetched

Instruction Fetch

- Fetch s instructions from I-cache
- I-Cache must be wide enough that each row of the I-Cache array can store s instructions and that an entire row can be accessed
- Fetch width = Row width
- Assume access latency is 1 cycle

Thank You