CS211
Advanced Computer Architecture

L12 Superscalar and VLIW

Chundong Wang

October 31st, 2025

Speculative Execution

- Branch Prediction
- Static
- Dynamic

Instruction-level parallelism

- How to achieve instruction-level parallelism?
  - Pipelining
  - Superscalar
  - Multiple processors
  - Multiple independent operations per instruction
    - VLIW

Superscalar

To fetch, issue to execution units, and complete more than one instruction at a time

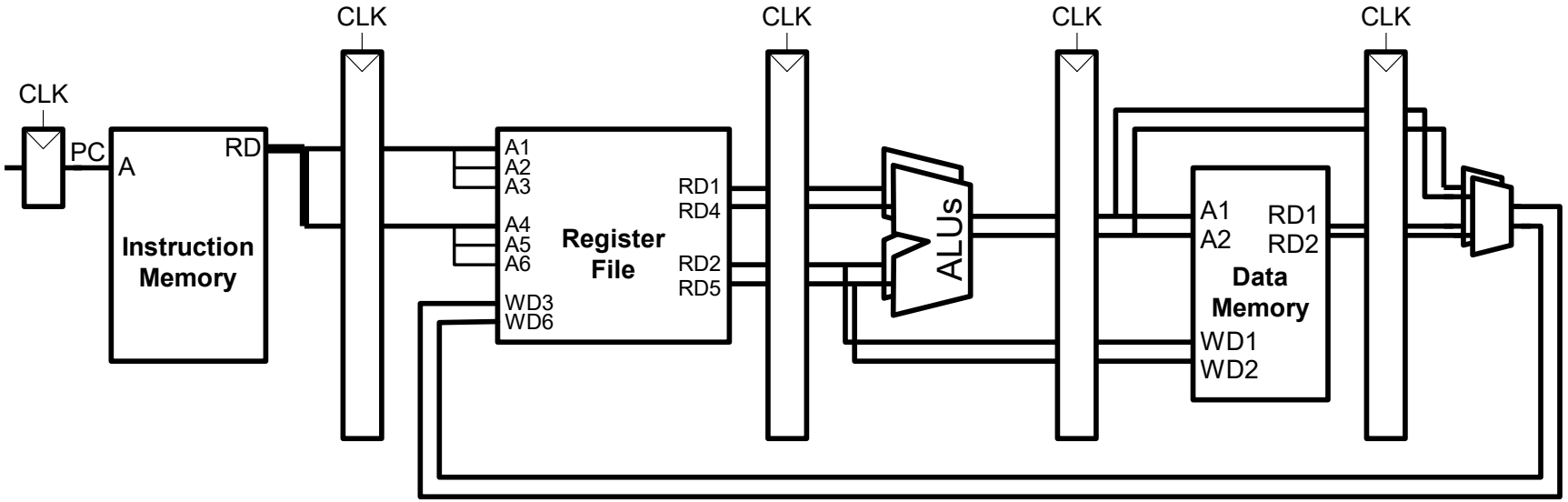
Superscalar Execution

- Idea: Fetch, decode, execute, retire multiple instructions per cycle
  - N-wide superscalar → N instructions per cycle
- Need to add the hardware resources for doing so
- Hardware performs the dependence checking between concurrently-fetched instructions
- Superscalar execution and out-of-order execution are orthogonal concepts
  - Can have all four combinations of processors:

    [in-order, out-of-order] x [scalar, superscalar]

In-Order Superscalar Processor Example

- Multiple copies of datapath: Can fetch/decode/execute multiple instructions per cycle
- **Dependencies** make it tricky to issue multiple instructions at once

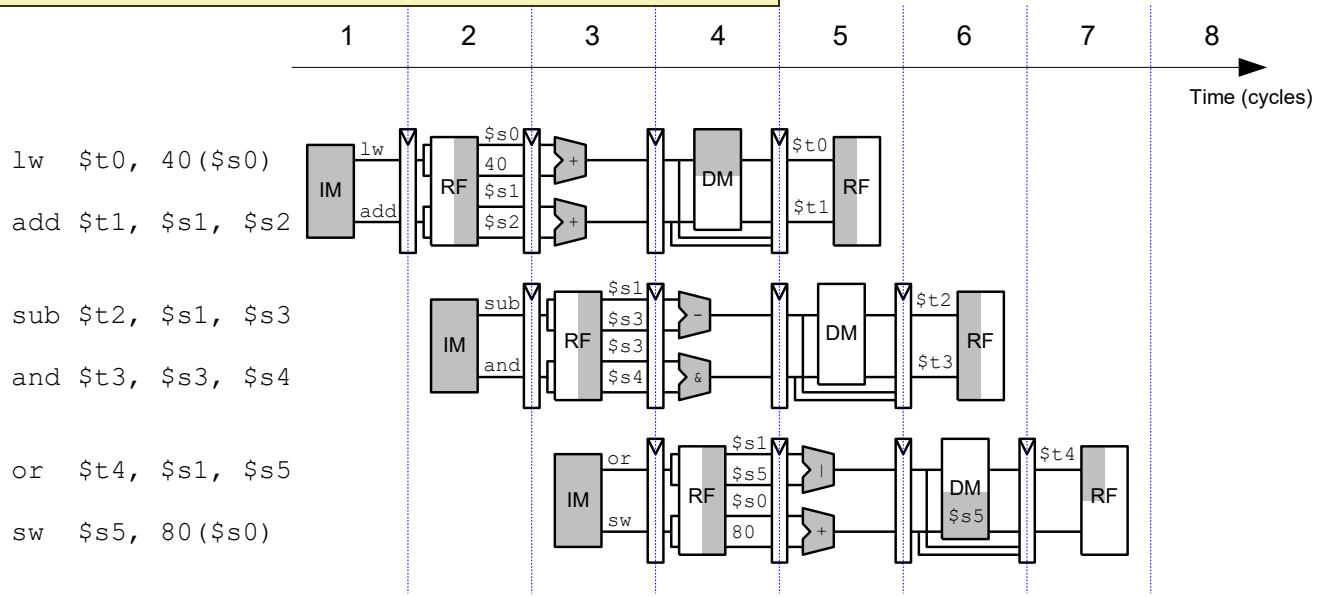


Here: Ideal IPC = 2

In-Order Superscalar Performance Example

```
lw  $t0, 40($s0)
add $t1, $s1, $s2
sub $t2, $s1, $s3
and $t3, $s3, $s4
or  $t4, $s1, $s5
sw  $s5, 80($s0)
```

Ideal IPC = 2

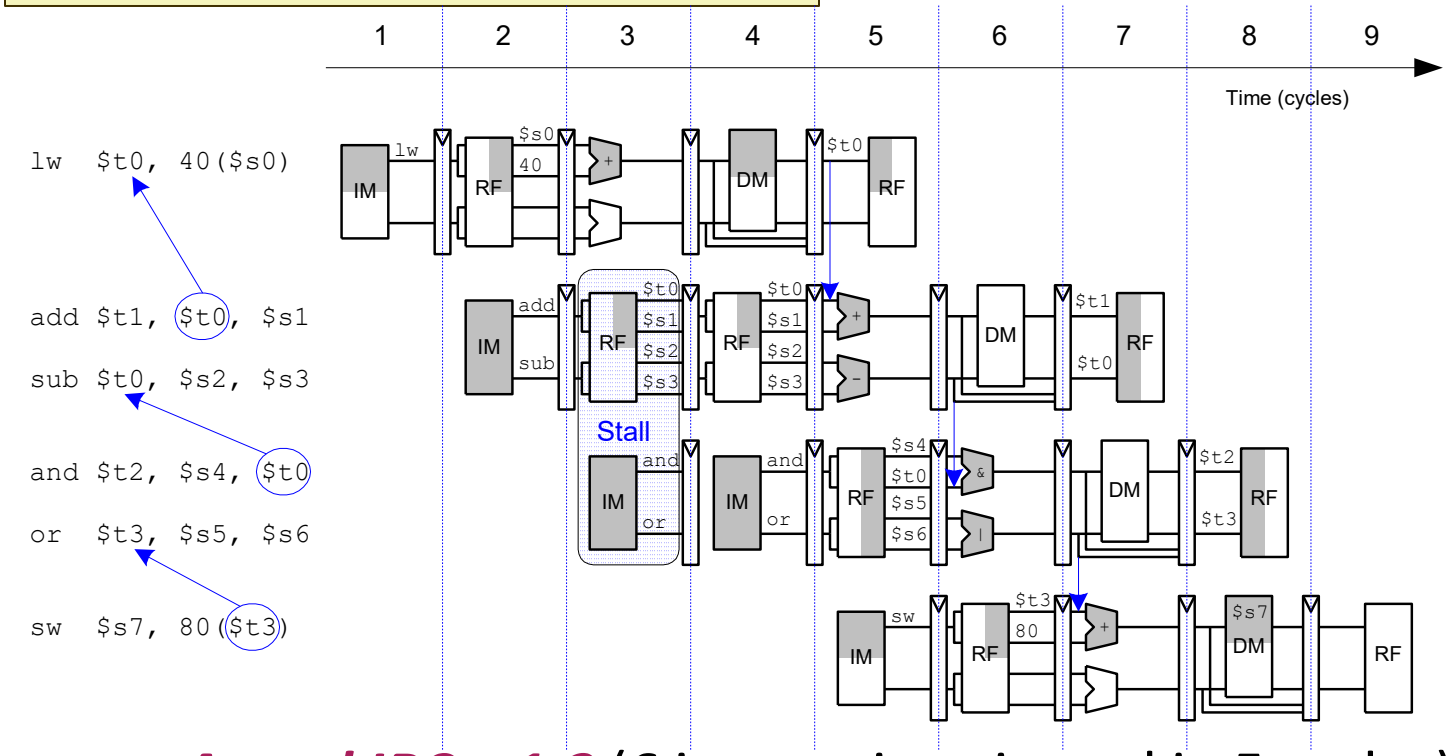


Actual IPC = 2 (6 instructions issued in 3 cycles)

Superscalar Performance with Dependencies

```
lw  $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or  $t3, $s5, $s6
sw  $s7, 80($t3)
```

Ideal IPC = 2



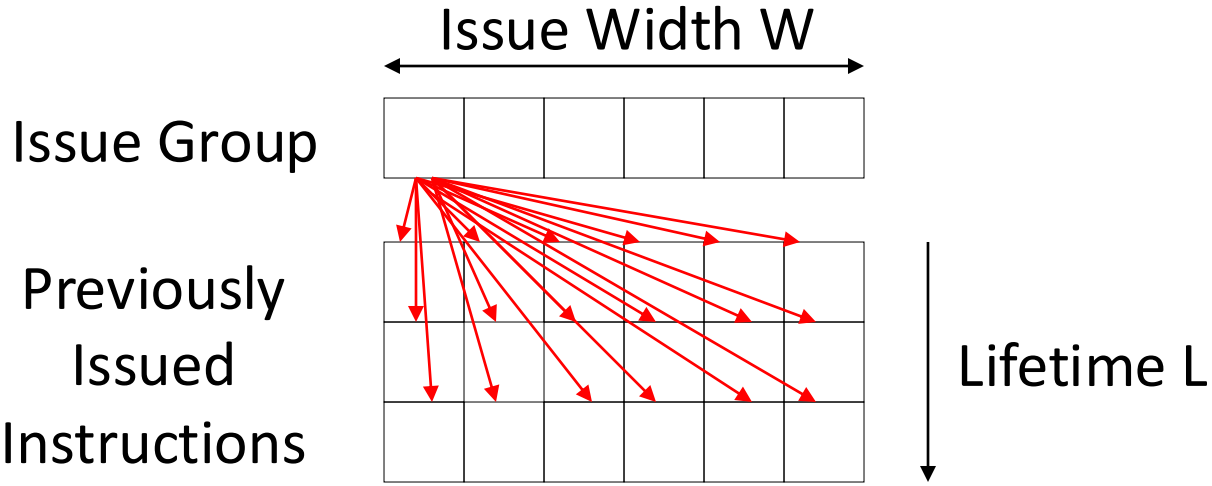
Actual IPC = 1.2 (6 instructions issued in 5 cycles)

Superscalar Execution Tradeoffs

- Advantages
  - Higher IPC (instructions per cycle)

- Disadvantages
  - Higher complexity for dependency checking
    - Require checking within a pipeline stage
    - Renaming becomes more complex in an OoO processor
  - More hardware resources needed

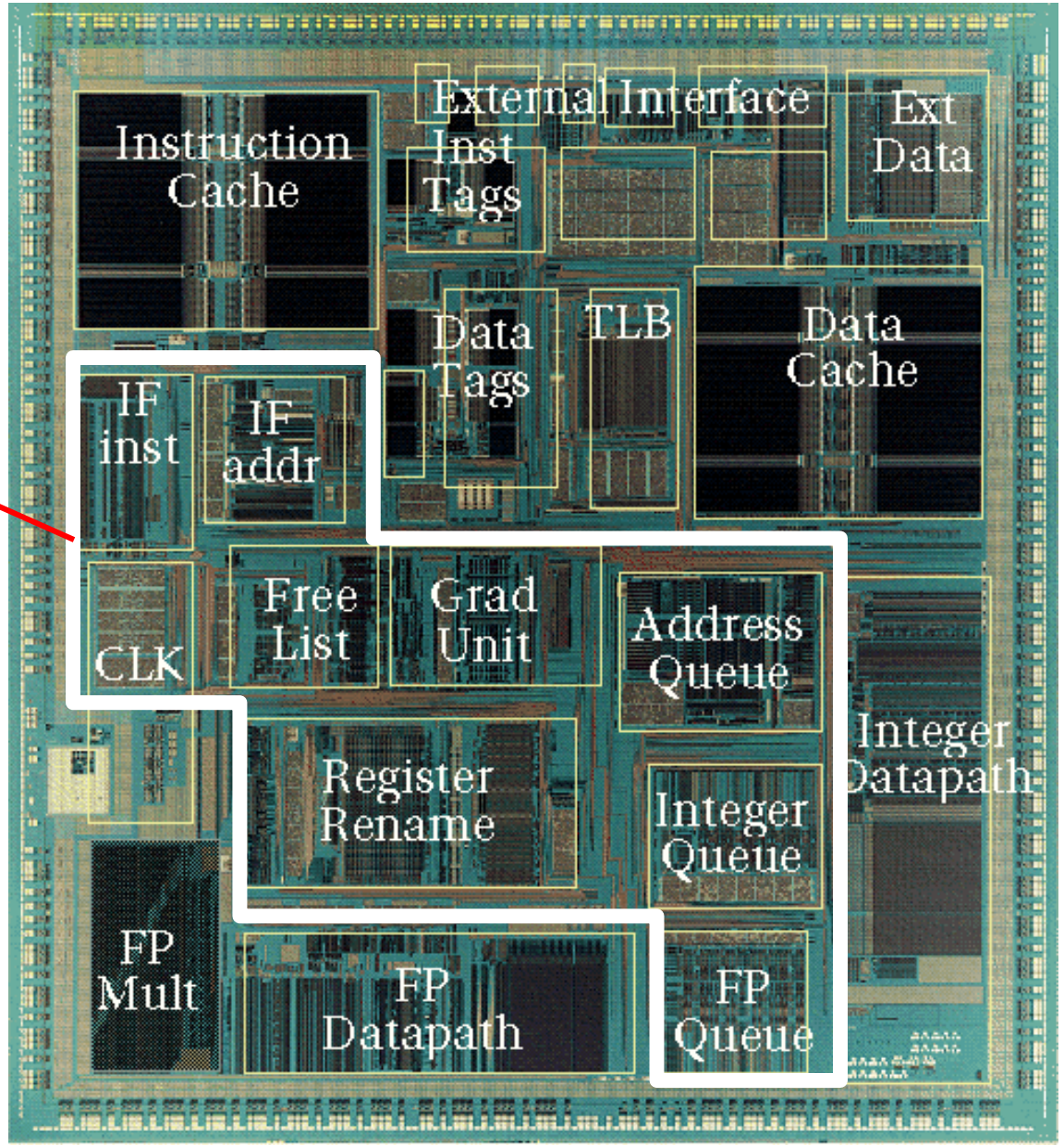
Superscalar Control Logic Scaling



- Each issued instruction must somehow check against W*L instructions, i.e., growth in hardware $\propto$ W*(W*L)
- For in-order machines, L is related to pipeline latencies and check is done during issue (interlocks or scoreboard)
- For out-of-order machines, L also includes time spent in instruction buffers (instruction window or ROB), and check is done by broadcasting tags to waiting instructions at write back (completion)
- As W increases, larger instruction window is needed to find enough parallelism to keep machine busy => greater L

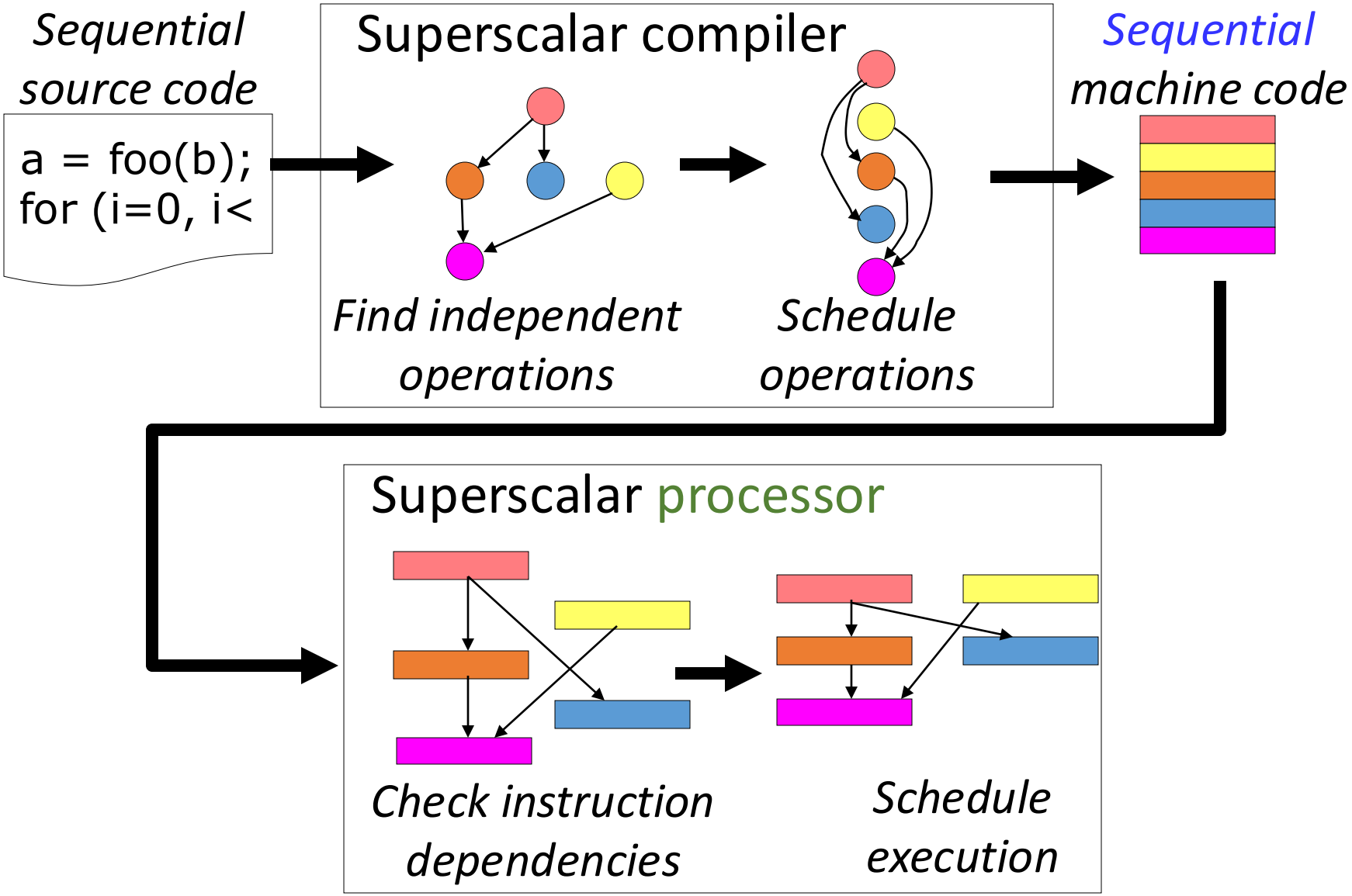
=> Out-of-order control logic grows faster than W^2 (~W^3)

Out-of-Order Control Complexity: MIPS R10000



[SGI/MIPS Technologies Inc., 1995]

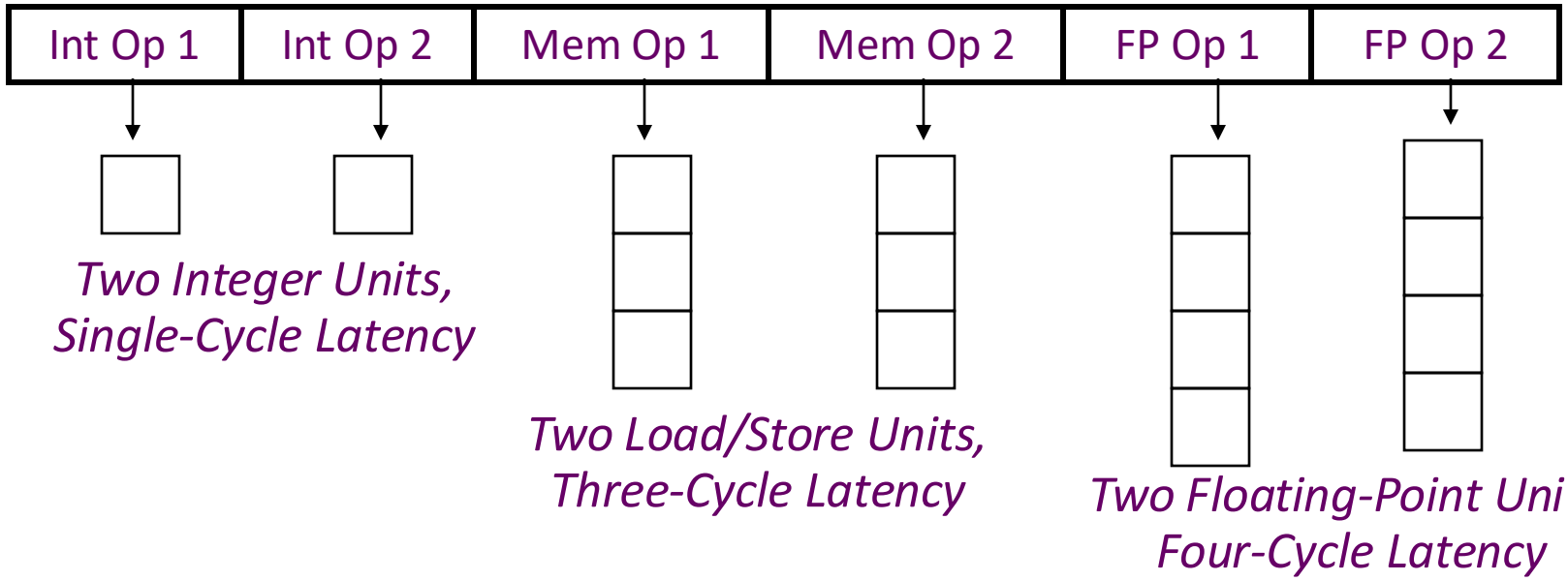
Sequential ISA Bottleneck



VLIW

Very Long Instruction Word

VLIW: Very Long Instruction Word



- Multiple operations packed into one instruction
- Each operation slot is for a fixed function
- Constant operation latencies are specified

VLIW Design Principles

- The architecture:
  - Allows operation parallelism within an instruction
    - No cross-operation RAW check
- Provides deterministic latency for all operations
  - Latency measured in ‘instructions’
  - No data use before data ready => no data interlocks

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

VLIW Compiler Responsibilities

- Schedule operations to maximize parallel execution
- Guarantee intra-instruction parallelism
- Schedule to avoid data hazards (no interlocks)
  - Typically separates operations with explicit NOPs

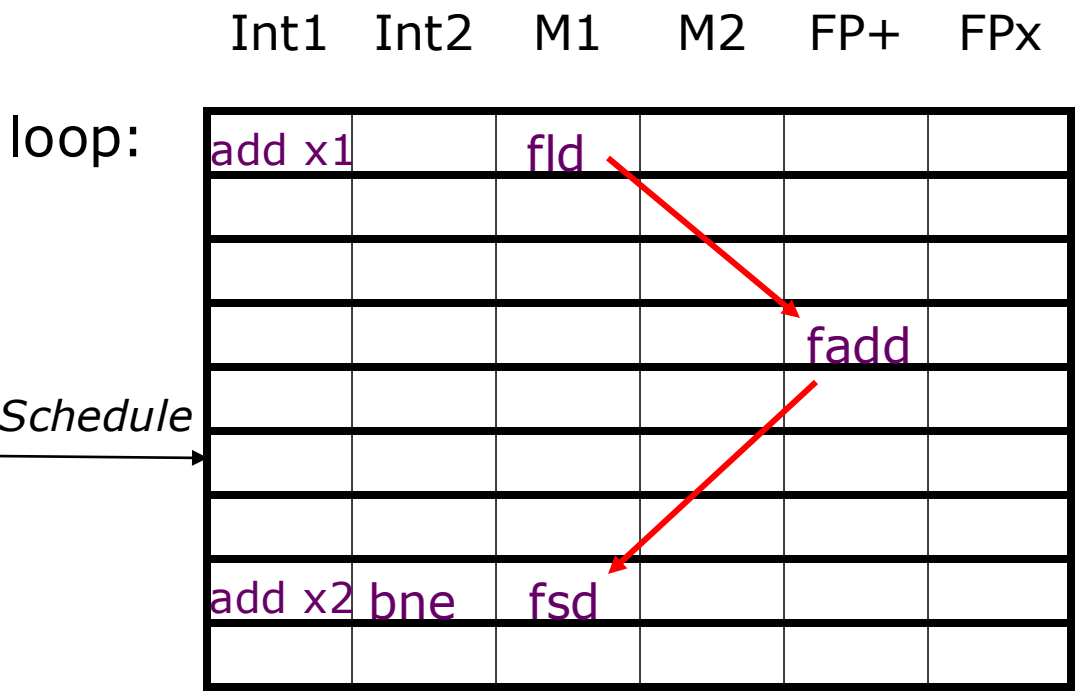
Loop Execution

```
for (i=0; i<N; i++)
    B[i] = A[i] + C;
```

↓ *Compile*

```
loop:   fld f1, 0(x1)
        add x1, 8
        fadd f2, f0, f1
        fsd f2, 0(x2)
        add x2, 8
        bne x1, x3, 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;
}
```

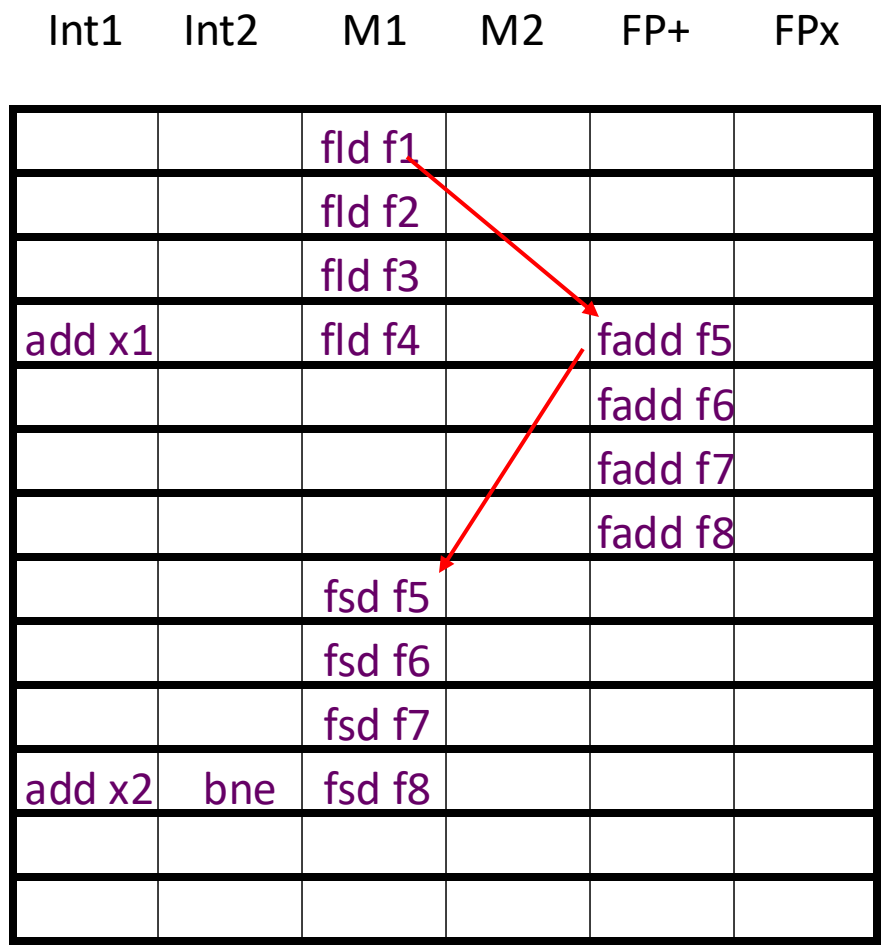
Need to handle values of N that are not multiples of unrolling factor with final cleanup loop

Scheduling Loop Unrolled Code

Unroll 4 ways

```
loop:  fld f1, 0(x1)
       fld f2, 8(x1)
       fld f3, 16(x1)
       fld f4, 24(x1)
       add x1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       fsd f5, 0(x2)
       fsd f6, 8(x2)
       fsd f7, 16(x2)
       fsd f8, 24(x2)
       add x2, 32
       bne x1, x3, loop
```

Schedule



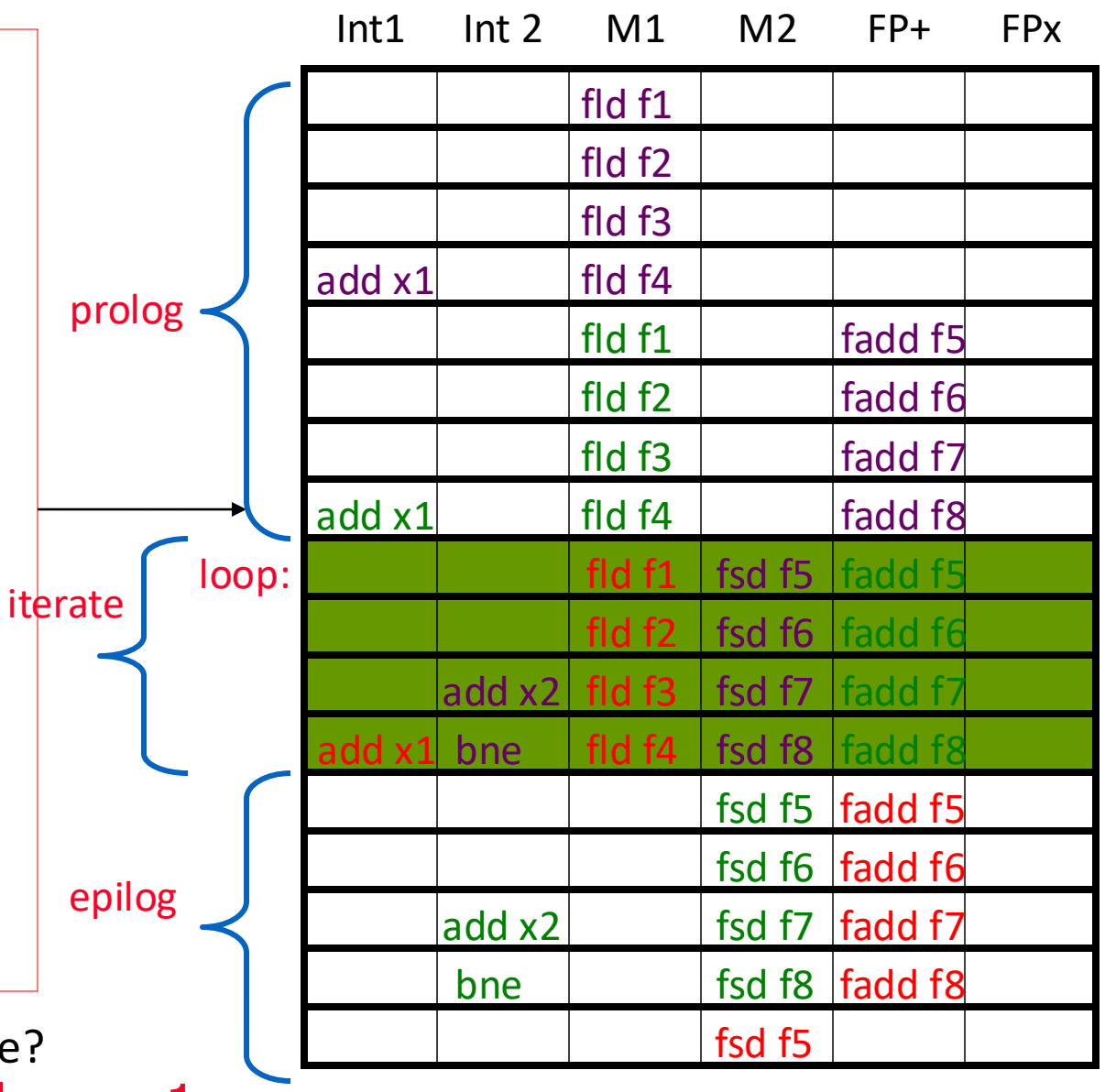
How many FLOPS/cycle? 4 fadds / 11 cycles = 0.36

Software Pipelining

上海科技大学 ShanghaiTech University

Unroll 4 ways first

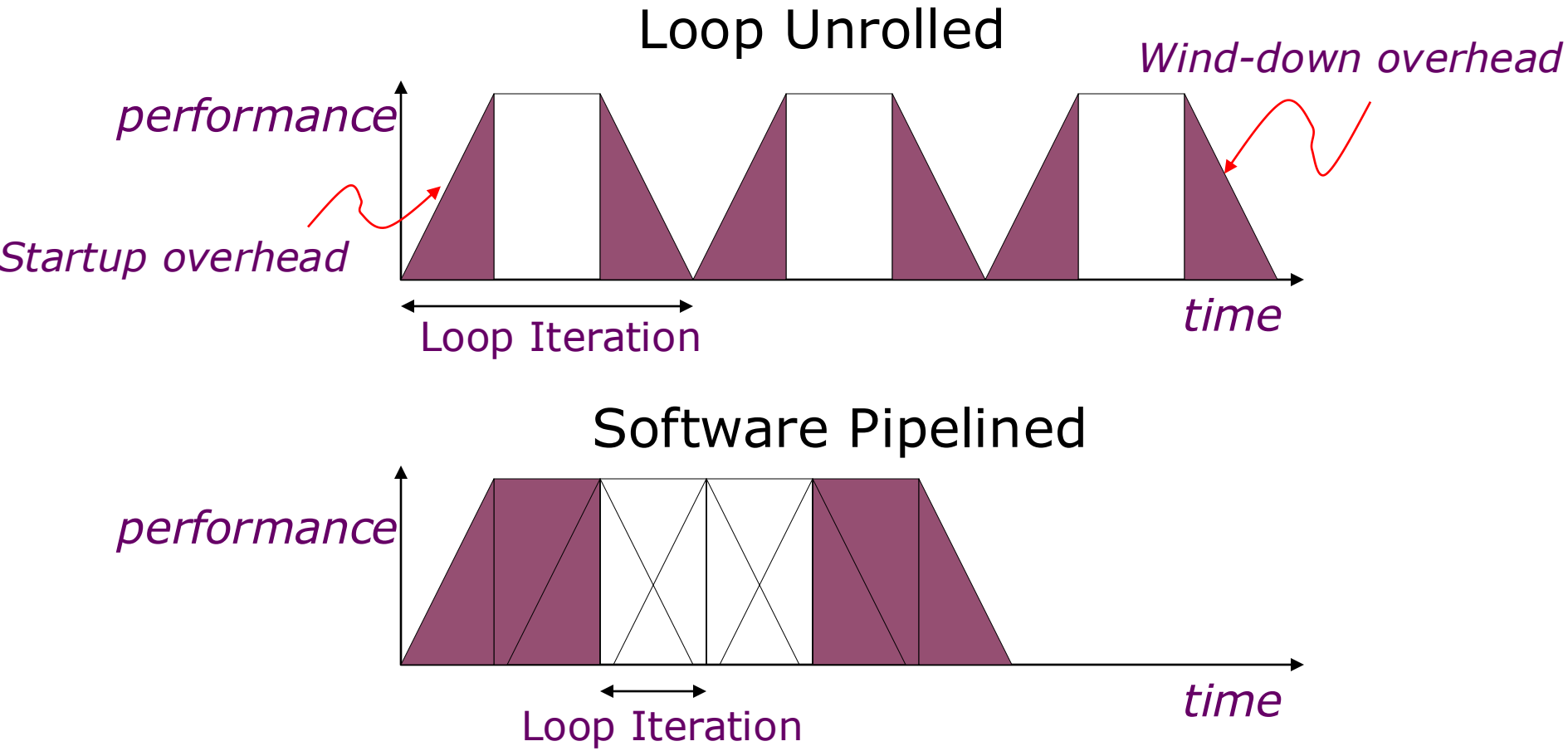
```
loop:  fld f1, 0(x1)
       fld f2, 8(x1)
       fld f3, 16(x1)
       fld f4, 24(x1)
       add x1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       fsd f5, 0(x2)
       fsd f6, 8(x2)
       fsd f7, 16(x2)
       add x2, 32
       fsd f8, -8(x2)
       bne x1, x3, loop
```



How many FLOPS/cycle?

4 fadds / 4 cycles = 1

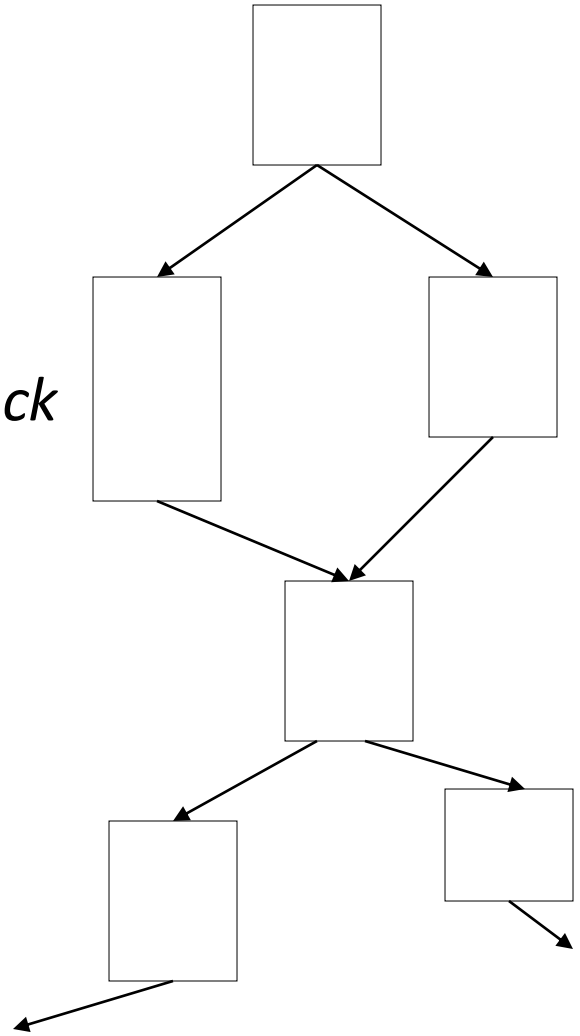
Software Pipelining vs. Loop Unrolling



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

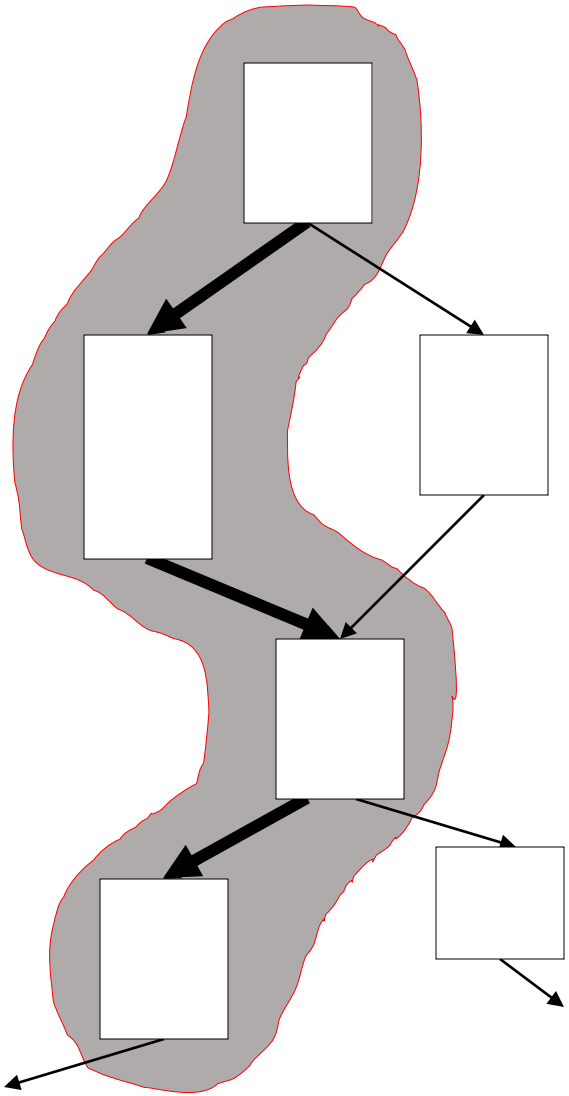
What if there are **no loops**?

Basic block



- Branches limit basic block size in control-flow intensive irregular code
- Difficult to find ILP in individual basic blocks

Trace Scheduling *[Fisher, Ellis]*



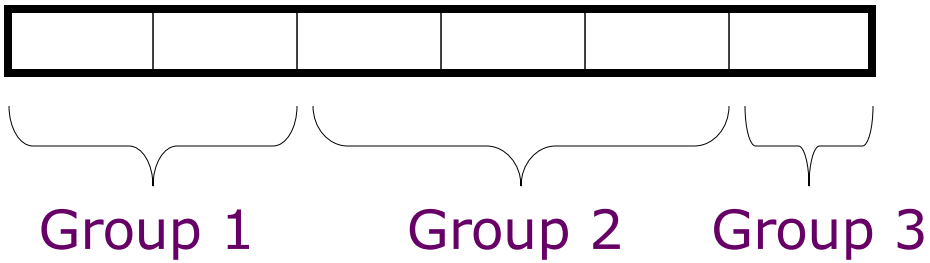
- Trace scheduling
  - A compiler technique that increases ILP by removing control dependencies, allowing operations following branches to be moved up and speculatively executed in parallel with operations before the branch.
  - *Trace selection*: find a likely sequence of basic blocks whose operations will be put together into a smaller number of instructions
  - *Trace compaction*: squeeze the trace into a small number of wide instructions
- Use profiling feedback or compiler heuristics to find common branch paths
- Schedule whole "trace" at once
- Add fix-up code to cope with branches jumping out of trace

Problems with “Classic” VLIW

- Knowing branch probabilities
  - Profiling requires a significant extra step in build process
- 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
- 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
  - Provide a single-op VLIW instruction
    - Cydra-5 UniOp instructions
  - Mark parallel groups
    - used in TMS320C6x DSPs, Intel IA-64



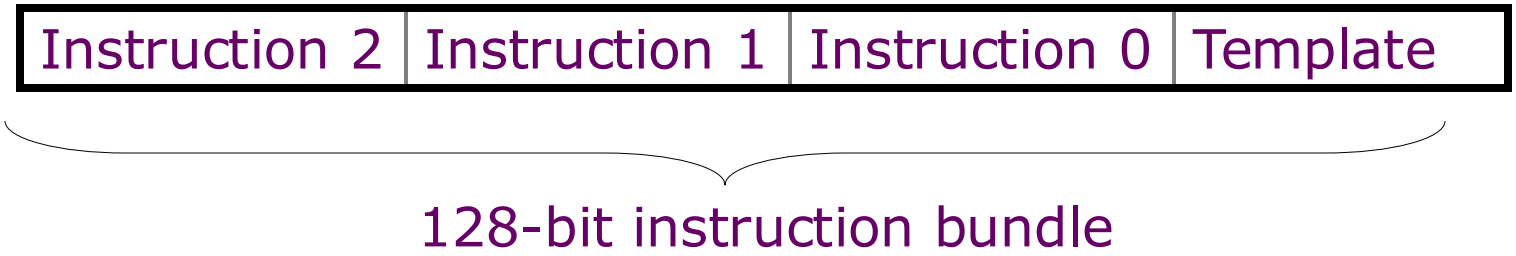
Cydra-5: Memory Latency Register (MLR)

- Problem: Loads have variable latency
- Solution: Let software choose desired memory latency
- Compiler schedules code for maximum load-use distance
- Software sets MLR to latency that matches code schedule
- Hardware ensures that loads take exactly MLR cycles to return values into processor pipeline
  - Hardware buffers loads that return early
  - Hardware stalls processor if loads return late

Intel Itanium, EPIC IA-64

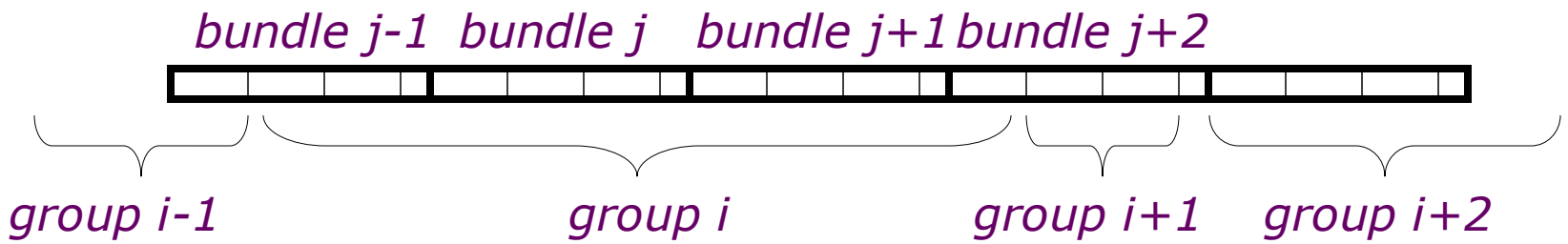
- EPIC is the style of architecture (cf. CISC, RISC)
  - Explicitly Parallel Instruction Computing (really just VLIW)
- IA-64 is Intel's chosen ISA (cf. x86, MIPS)
  - IA-64 = Intel Architecture 64-bit
  - An object-code-compatible VLIW
- Merced was first Itanium implementation (cf. 8086)
  - First customer shipment expected 1997 (actually 2001)
  - McKinley, second implementation shipped in 2002
  - Recent version, Poulson, eight cores, 32nm, announced 2011

IA-64 Instruction Format



IA-64 instructions are encoded in bundles.

- Template bits (5 bits) describe grouping of these instructions (3*41bits) with others in adjacent bundles
- Each group contains instructions that can execute in parallel



IA-64 Registers

- 128 General Purpose 64-bit Integer Registers
- 128 General Purpose 64/80-bit Floating Point Registers
- 64 1-bit Predicate (*not Predict*) Registers
  - Used to support branch predication (*not prediction*)
  - Encoded and placed in the lower 6 bits of each instruction
  - Set using compare or test instructions
    - If predicate register set, run that instruction
    - Otherwise, instruction treated as nop

IA-64 Registers

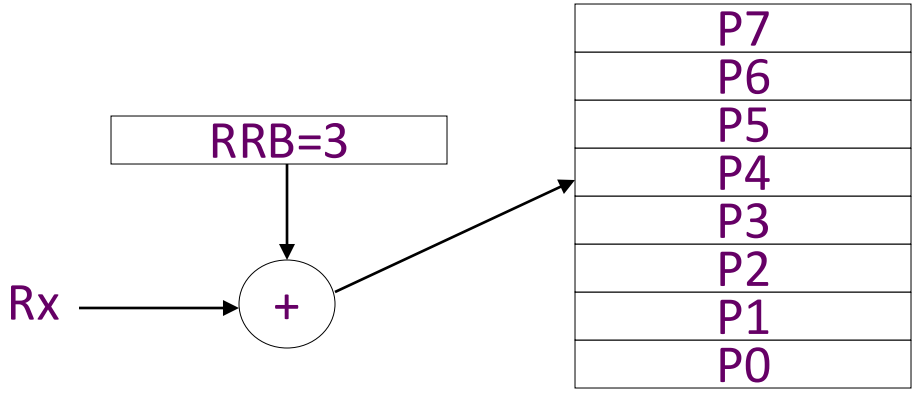
- 128 General Purpose 64-bit Integer Registers
- 128 General Purpose 64/80-bit Floating Point Registers
- 64 1-bit Predicate Registers

- GPRs "rotate" to reduce code size for software pipelined loops
  - Rotation is a simple form of register renaming allowing one instruction to address different physical registers on each iteration
  - To support software pipelining

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

  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.

R0 to R31 are static and refer to the first 32 physical registers;
R32 to R127 are rotating registers

Rotating Register File
(Previous Loop Example)

Three cycle load latency encoded as difference of 3 in register specifier number (f4 - f1 = 3)

Four cycle fadd latency encoded as difference of 4 in register specifier number (f9 – f5 = 4)

ld f1, ()	fadd f5, f4, ...	sd f9, ()	bloop

ld P9, ()	fadd P13, P12,	sd P17, ()	bloop	RRB=8
ld P8, ()	fadd P12, P11,	sd P16, ()	bloop	RRB=7
ld P7, ()	fadd P11, P10,	sd P15, ()	bloop	RRB=6
ld P6, ()	fadd P10, P9,	sd P14, ()	bloop	RRB=5
ld P5, ()	fadd P9, P8,	sd P13, ()	bloop	RRB=4
ld P4, ()	fadd P8, P7,	sd P12, ()	bloop	RRB=3
ld P3, ()	fadd P7, P6,	sd P11, ()	bloop	RRB=2
ld P2, ()	fadd P6, P5,	sd P10, ()	bloop	RRB=1

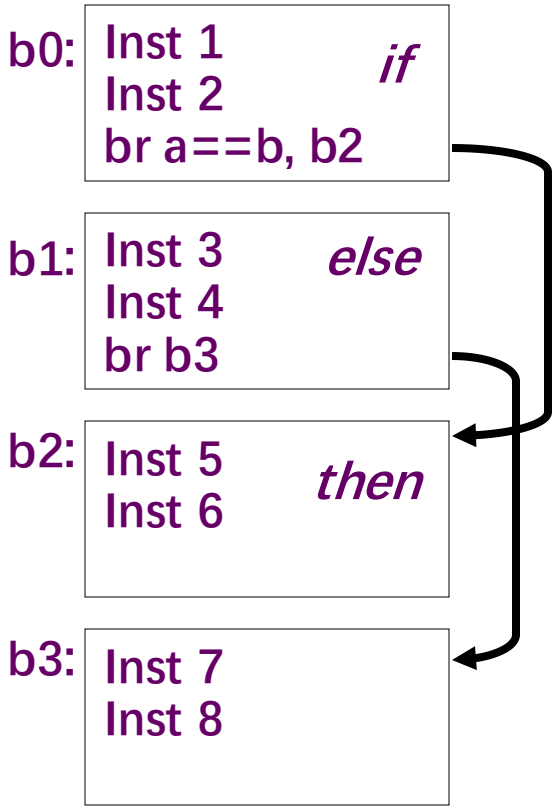
IA-64 Predicated Execution

上海科技大学 ShanghaiTech University

Problem: Mispredicted branches limit ILP

Solution: Eliminate hard to predict branches with predicated execution

- Almost all IA-64 instructions can be executed conditionally under predicate
- Instruction becomes NOP if predicate register false



Four basic blocks

Predication

```
Inst 1
Inst 2
p1,p2 <- cmp(a==b)
(p1) Inst 3     ||   (p2) Inst 5
(p1) Inst 4     ||   (p2) Inst 6
Inst 7
Inst 8
```

One basic block

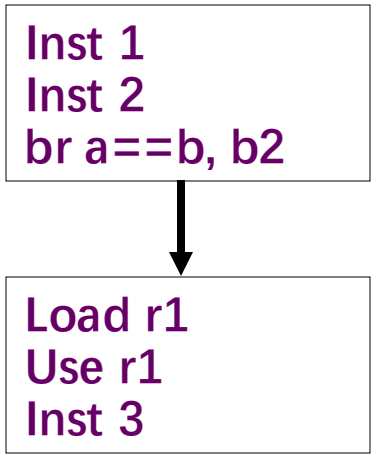
Mahlke et al, ISCA95: On average >50% branches removed

Warning: Complicates bypassing!

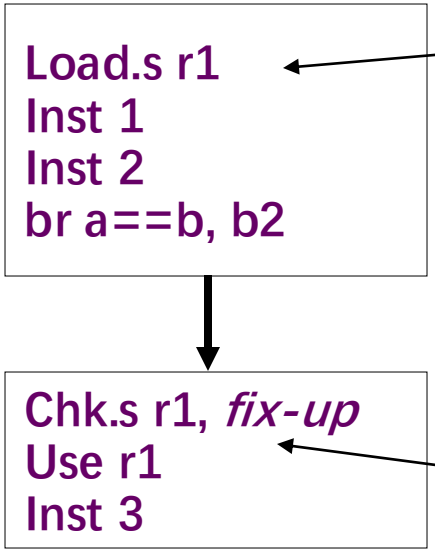
IA-64 Speculative Execution

Problem: Branches restrict compiler code motion

Solution: Speculative operations that don't cause exceptions



Can't move load above branch because might cause spurious exception



Speculative load never causes exception, but sets "poison" bit on destination register

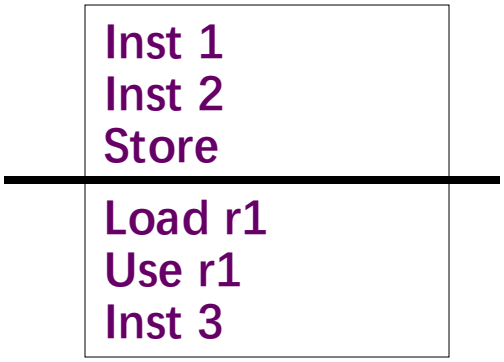
Check for exception in original home block jumps to fix-up code if exception detected

Particularly useful for scheduling long latency loads early

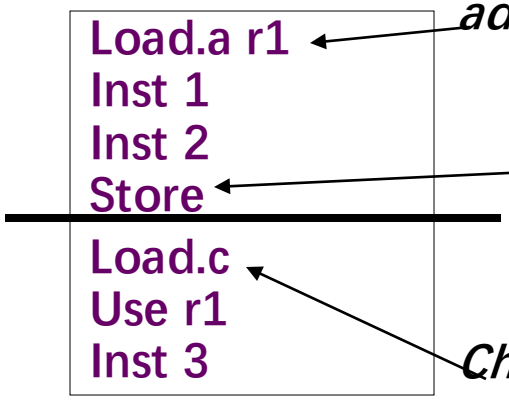
IA-64 Data Speculation

Problem: Possible memory hazards limit code scheduling

Solution: Hardware to check pointer hazards



Can't move load above store because store might be to same address



Data speculative load adds address to address check table

Store invalidates any matching loads in address check table

Check if load invalid (or missing), jump to fixup code if so

Requires associative hardware in address check table

Limits of Static Scheduling

- Statically unpredictable branches
- Variable memory latency (unpredictable cache misses)
- Code size explosion
- Compiler complexity
- Despite several attempts, VLIW has failed in general-purpose computing arena (so far).
  - More complex VLIW architectures are close to in-order superscalar in complexity, no real advantage on large complex apps.
- Successful in embedded DSP market
  - Simpler VLIWs with more constrained environment, friendlier code.

Intel Kills Itanium

- Donald Knuth " … *Itanium approach that was supposed to be so terrific—until it turned out that the wished-for compilers were basically impossible to write*."

- "*Intel officially announced the end of life and product discontinuance of the Itanium CPU family on January 30th, 2019*", Wikipedia

Conclusion

- Superscalar
- VLIW

Acknowledgements

- These slides contain materials developed and copyright by:
  - Prof. Krste Asanovic (UC Berkeley)
  - Prof. Joel Emer (MIT)
  - Prof. Daniel Sanchez (MIT)
  - Prof. Onur Mutlu (ETHZ)