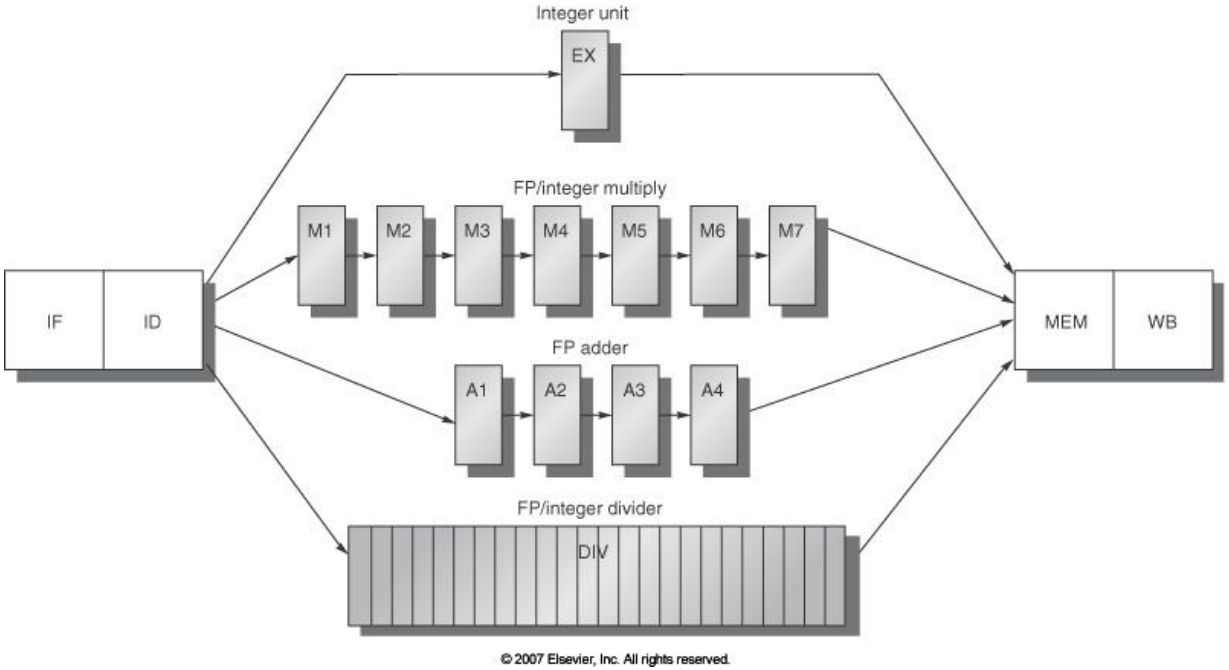
Lecture 5: Pipeline Wrap-up, Static ILP

- Topics: multi-cycle ops, precise interrupts, compiler scheduling, loop unrolling, software pipelining (Sections C.5, 3.2)

- Please hand in Assignment 1 now

Multicycle Instructions



Functional unit	Latency	Initiation interval
Integer ALU	1	1
Data memory	2	1
FP add	4	1
FP multiply	7	1
FP divide	25	25

Effects of Multicycle Instructions

- Structural hazards if the unit is not fully pipelined (divider)
- Frequent RAW hazard stalls
- Potentially multiple writes to the register file in a cycle
- WAW hazards because of out-of-order instr completion
- Imprecise exceptions because of o-o-o instr completion

Note: Can also increase the “width” of the processor: handle multiple instructions at the same time: for example, fetch two instructions, read registers for both, execute both, etc.

Precise Exceptions

- On an exception:
  - must save PC of instruction where program must resume
  - all instructions after that PC that might be in the pipeline must be converted to NOPs (other instructions continue to execute and may raise exceptions of their own)
  - temporary program state not in memory (in other words, registers) has to be stored in memory
  - potential problems if a later instruction has already modified memory or registers

- A processor that fulfils all the above conditions is said to provide precise exceptions (useful for debugging and of course, correctness)

Dealing with these Effects

- Multiple writes to the register file: increase the number of ports, stall one of the writers during ID, stall one of the writers during WB (the stall will propagate)
- WAW hazards: detect the hazard during ID and stall the later instruction
- Imprecise exceptions: buffer the results if they complete early or save more pipeline state so that you can return to exactly the same state that you left at

ILP

- Instruction-level parallelism: overlap among instructions: pipelining or multiple instruction execution

- What determines the degree of ILP?
  - dependences: property of the program
  - hazards: property of the pipeline

Static vs Dynamic Scheduling

- Arguments against dynamic scheduling:
  - requires complex structures to identify independent instructions (scoreboards, issue queue)
    - high power consumption
    - low clock speed
    - high design and verification effort
  - the compiler can “easily” compute instruction latencies and dependences – complex software is always preferred to complex hardware (?)

Loop Scheduling

- Revert back to the 5-stage in-order pipeline
- The compiler's job is to minimize stalls
- Focus on loops: account for most cycles, relatively easy to analyze and optimize
- Recall: a load has a two-cycle latency (1 stall cycle for the consumer that immediately follows), FP ALU feeding another → 3 stall cycles, FP ALU feeding a store → 2 stall cycles, int ALU feeding a branch → 1 stall cycle, one delay slot after a branch

Loop Example

```
for (i=1000; i>0; i--)
    x[i] = x[i] + s;
```

Source code

```
Loop:   L.D      F0, 0(R1)       ; F0 = array element
        ADD.D    F4, F0, F2      ; add scalar
        S.D      F4, 0(R1)       ; store result
        DADDUI   R1, R1,# -8     ; decrement address pointer
        BNE      R1, R2, Loop    ; branch if R1 != R2
        NOP
```

Assembly code

Loop Example

```
for (i=1000; i>0; i--)
    x[i] = x[i] + s;
```

Source code

```
Loop:   L.D      F0, 0(R1)      ; F0 = array element
        ADD.D    F4, F0, F2     ; add scalar
        S.D      F4, 0(R1)      ; store result
        DADDUI   R1, R1,# -8    ; decrement address pointer
        BNE      R1, R2, Loop   ; branch if R1 != R2
        NOP
```

Assembly code

```
Loop:   L.D      F0, 0(R1)      ; F0 = array element
        stall
        ADD.D    F4, F0, F2     ; add scalar
        stall
        stall
        S.D      F4, 0(R1)      ; store result
        DADDUI   R1, R1,# -8    ; decrement address pointer
        stall
        BNE      R1, R2, Loop   ; branch if R1 != R2
        stall
```

10-cycle schedule

Smart Schedule

```
Loop:   L.D      F0, 0(R1)
        stall
        ADD.D    F4, F0, F2
        stall
        stall
        S.D      F4, 0(R1)
        DADDUI   R1, R1,# -8
        stall
        BNE      R1, R2, Loop
        stall
```

```
Loop:   L.D      F0, 0(R1)
        DADDUI   R1, R1,# -8
        ADD.D    F4, F0, F2
        stall
        BNE      R1, R2, Loop
        S.D      F4, 8(R1)
```

- By re-ordering instructions, it takes 6 cycles per iteration instead of 10
- We were able to violate an anti-dependence easily because an immediate was involved
- Loop overhead (instrs that do book-keeping for the loop): 2
  Actual work (the ld, add.d, and s.d): 3 instrs
  Can we somehow get execution time to be 3 cycles per iteration?

Loop Unrolling

```
Loop:   L.D      F0, 0(R1)
        ADD.D    F4, F0, F2
        S.D      F4, 0(R1)
        L.D      F6, -8(R1)
        ADD.D    F8, F6, F2
        S.D      F8, -8(R1)
        L.D      F10,-16(R1)
        ADD.D    F12, F10, F2
        S.D      F12, -16(R1)
        L.D      F14, -24(R1)
        ADD.D    F16, F14, F2
        S.D      F16, -24(R1)
        DADDUI   R1, R1, #-32
        BNE      R1,R2, Loop
```

- Loop overhead: 2 instrs; Work: 12 instrs
- How long will the above schedule take to complete?

Scheduled and Unrolled Loop

```
Loop:   L.D      F0, 0(R1)
        L.D      F6, -8(R1)
        L.D      F10,-16(R1)
        L.D      F14, -24(R1)
        ADD.D    F4, F0, F2
        ADD.D    F8, F6, F2
        ADD.D    F12, F10, F2
        ADD.D    F16, F14, F2
        S.D      F4, 0(R1)
        S.D      F8, -8(R1)
        DADDUI   R1, R1, # -32
        S.D      F12, 16(R1)
        BNE      R1,R2, Loop
        S.D      F16, 8(R1)
```

- Execution time: 14 cycles or 3.5 cycles per original iteration

Loop Unrolling

- Increases program size
- Requires more registers
- To unroll an n-iteration loop by degree k, we will need (n/k) iterations of the larger loop, followed by (n mod k) iterations of the original loop

Automating Loop Unrolling

- Determine the dependences across iterations: in the example, we knew that loads and stores in different iterations did not conflict and could be re-ordered
- Determine if unrolling will help – possible only if iterations are independent
- Determine address offsets for different loads/stores
- Dependency analysis to schedule code without introducing hazards; eliminate name dependences by using additional registers

Superscalar Pipelines

Integer pipeline	FP pipeline
Handles L.D, S.D, ADDUI, BNE	Handles ADD.D

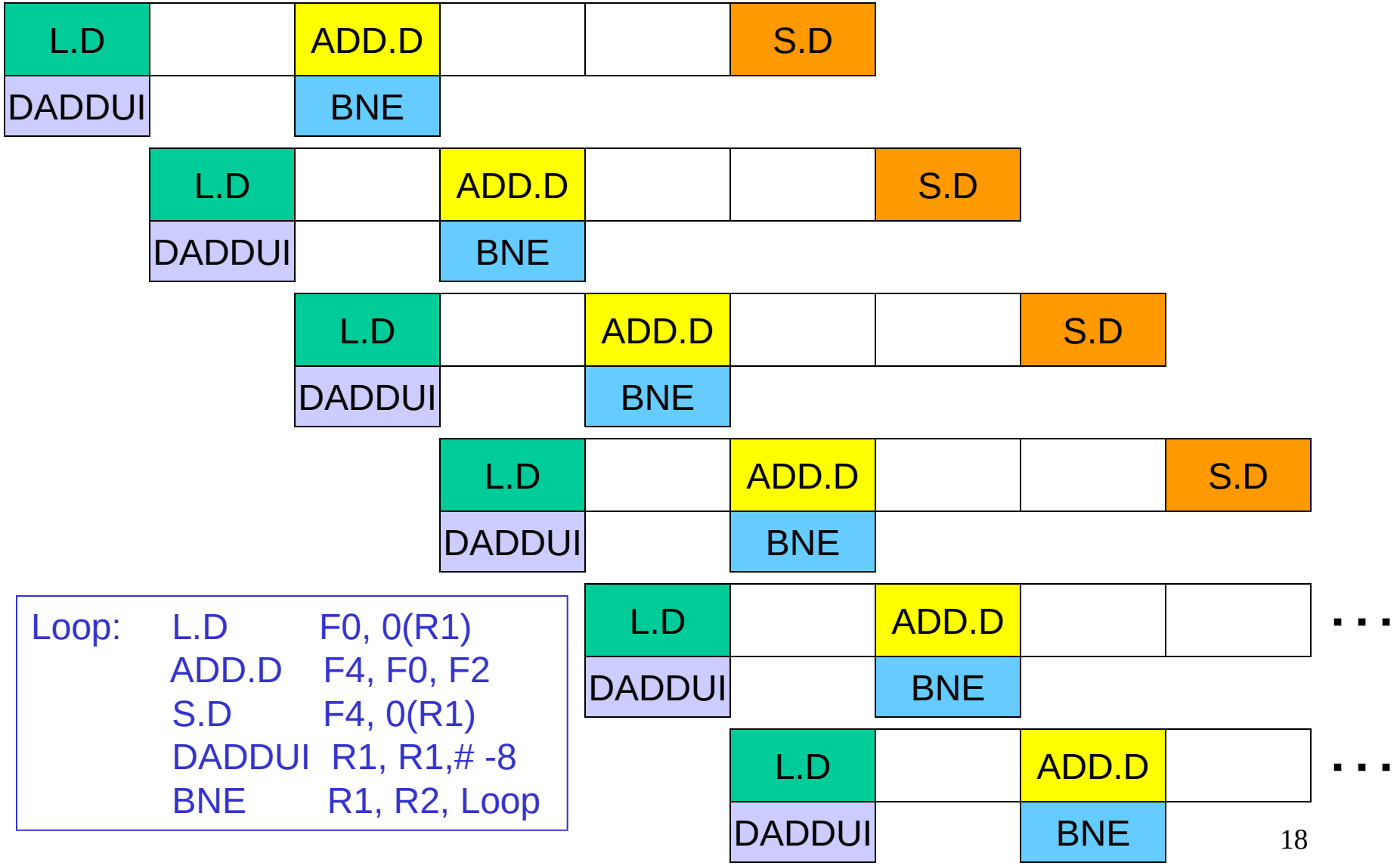
- What is the schedule with an unroll degree of 4?

Superscalar Pipelines

	Integer pipeline		FP pipeline	
Loop:	L.D	F0,0(R1)		
	L.D	F6,-8(R1)		
	L.D	F10,-16(R1)	ADD.D	F4,F0,F2
	L.D	F14,-24(R1)	ADD.D	F8,F6,F2
	L.D	F18,-32(R1)	ADD.D	F12,F10,F2
	S.D	F4,0(R1)	ADD.D	F16,F14,F2
	S.D	F8,-8(R1)	ADD.D	F20,F18,F2
	S.D	F12,-16(R1)		
	DADDUI	R1,R1,# -40		
	S.D	F16,16(R1)		
	BNE	R1,R2,Loop		
	S.D	F20,8(R1)		

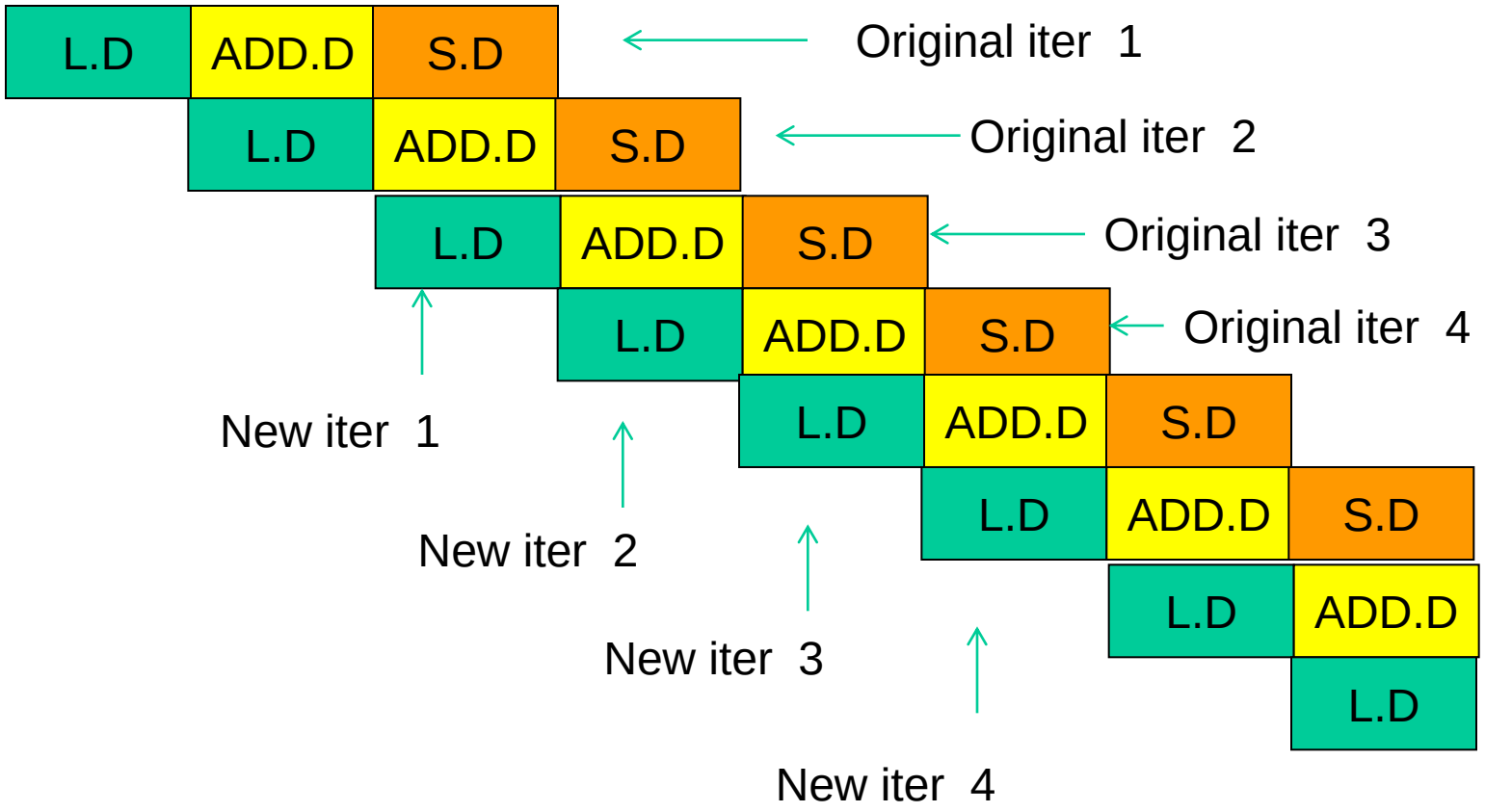
- Need unroll by degree 5 to eliminate stalls
- The compiler may specify instructions that can be issued as one packet
- The compiler may specify a fixed number of instructions in each packet: Very Large Instruction Word (VLIW)

Software Pipeline?!




```
Loop:   L.D     F0, 0(R1)
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)
        DADDUI  R1, R1,# -8
        BNE     R1, R2, Loop
```

Software Pipeline



Software Pipelining

```
Loop:   L.D      F0, 0(R1)
        ADD.D    F4, F0, F2
        S.D      F4, 0(R1)
        DADDUI   R1, R1,# -8
        BNE      R1, R2, Loop
```

```
Loop:   S.D      F4, 16(R1)
        ADD.D    F4, F0, F2
        L.D      F0, 0(R1)
        DADDUI   R1, R1,# -8
        BNE      R1, R2, Loop
```

- Advantages: achieves nearly the same effect as loop unrolling, but without the code expansion – an unrolled loop may have inefficiencies at the start and end of each iteration, while a sw-pipelined loop is almost always in steady state – a sw-pipelined loop can also be unrolled to reduce loop overhead

- Disadvantages: does not reduce loop overhead, may require more registers

Title

- Bullet