250P: Computer Systems Architecture

Lecture 7: Static Instruction Level Parallelism

Anton Burtsev

April, 2020

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 (?)

ILP

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

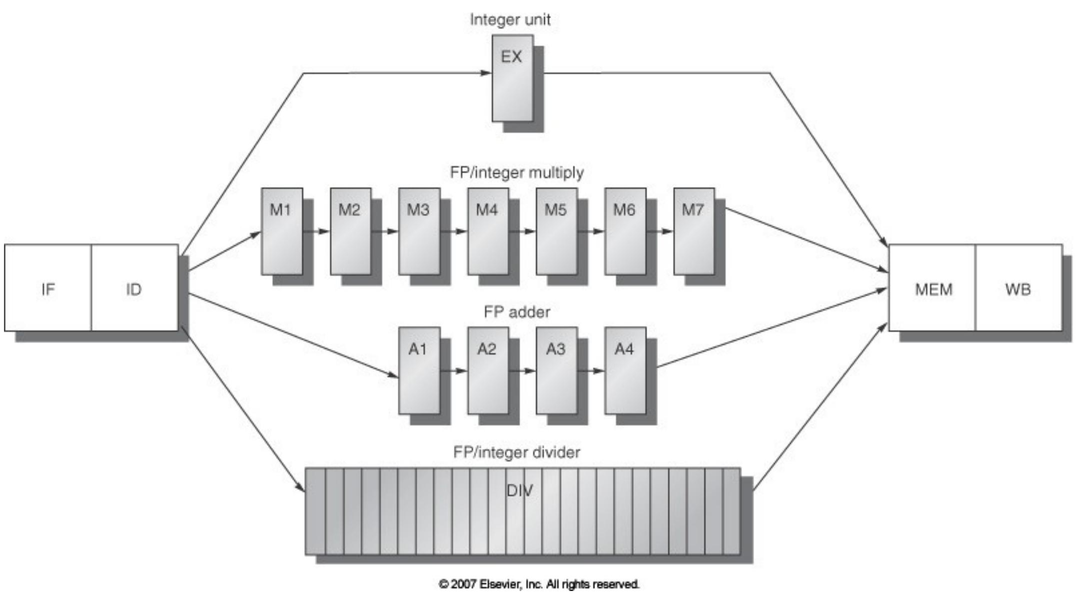
- What determines the degree of ILP?
  - dependences: property of the program
  - hazards: property of the pipeline

Loop Scheduling

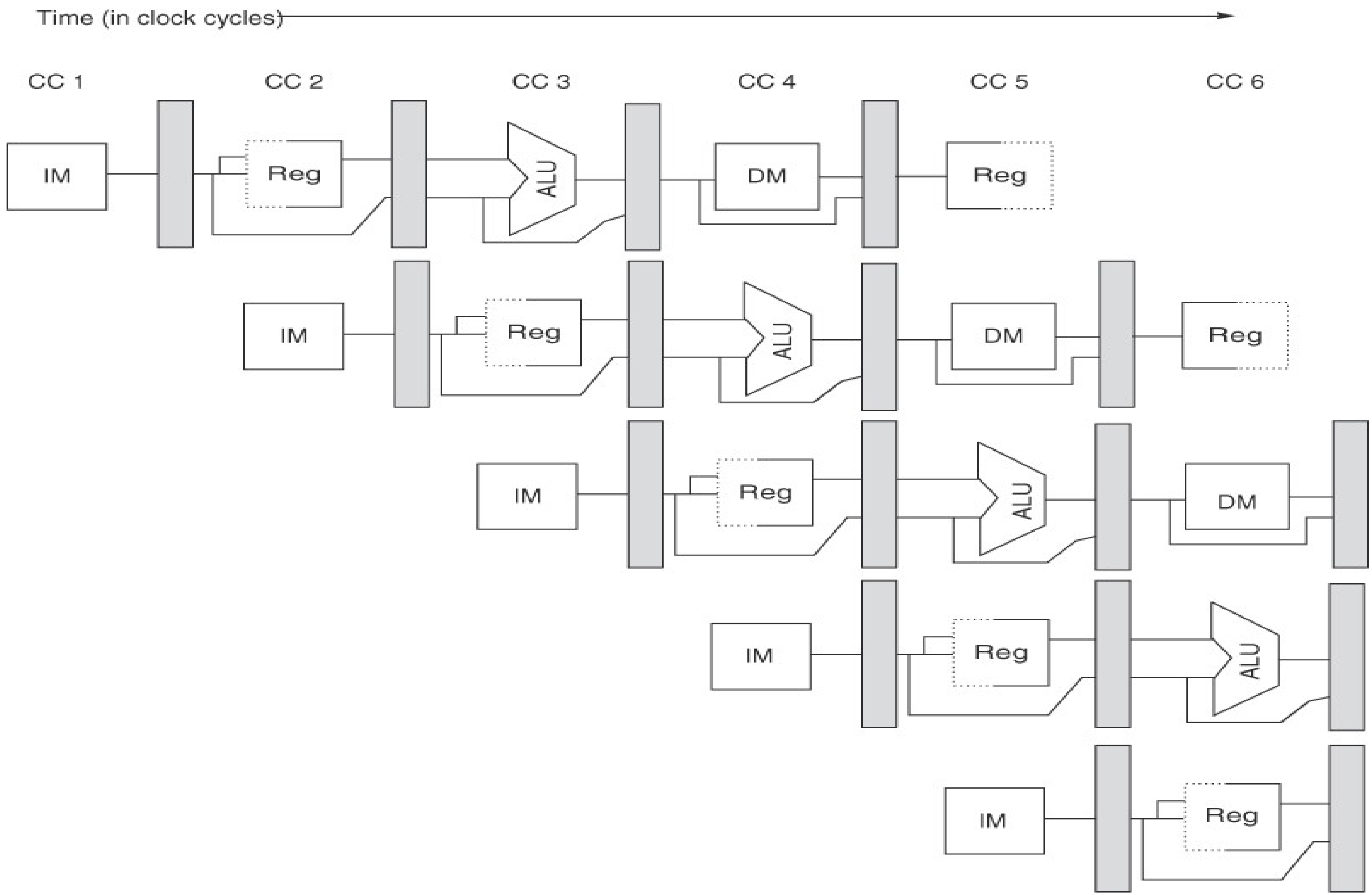
- The compiler's job is to minimize stalls
- Focus on loops: account for most cycles, relatively easy to analyze and optimize

Assumptions

- Load: 2-cycles (1 cycle stall for consumer)
- FP ALU: 4-cycles (3 cycle stall for consumer; 2 cycle stall if the consumer is a store)
- One branch delay slot
- Int ALU: 1-cycle (no stall for consumer, 1 cycle stall if the consumer is a branch)



LD -> any : 1 stall
FPALU -> any: 3 stalls
FPALU -> ST : 2 stalls
IntALU -> BR : 1 stall



Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
IM
Reg
ALU
DM
Reg
IM
Reg
ALU
DM
Reg
IM
Reg
ALU
DM
IM
Reg
ALU
IM
Reg

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

LD -> any : 1 stall
FPALU -> any: 3 stalls
FPALU -> ST : 2 stalls
IntALU -> BR : 1 stall

```
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

LD -> any : 1 stall
FPALU -> any: 3 stalls
FPALU -> ST : 2 stalls
IntALU -> BR : 1 stall

```
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
```

LD -> any : 1 stall
FPALU -> any: 3 stalls
FPALU -> ST : 2 stalls
IntALU -> BR : 1 stall

- 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)
```

LD -> any : 1 stall
FPALU -> any: 3 stalls
FPALU -> ST : 2 stalls
IntALU -> BR : 1 stall

- 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

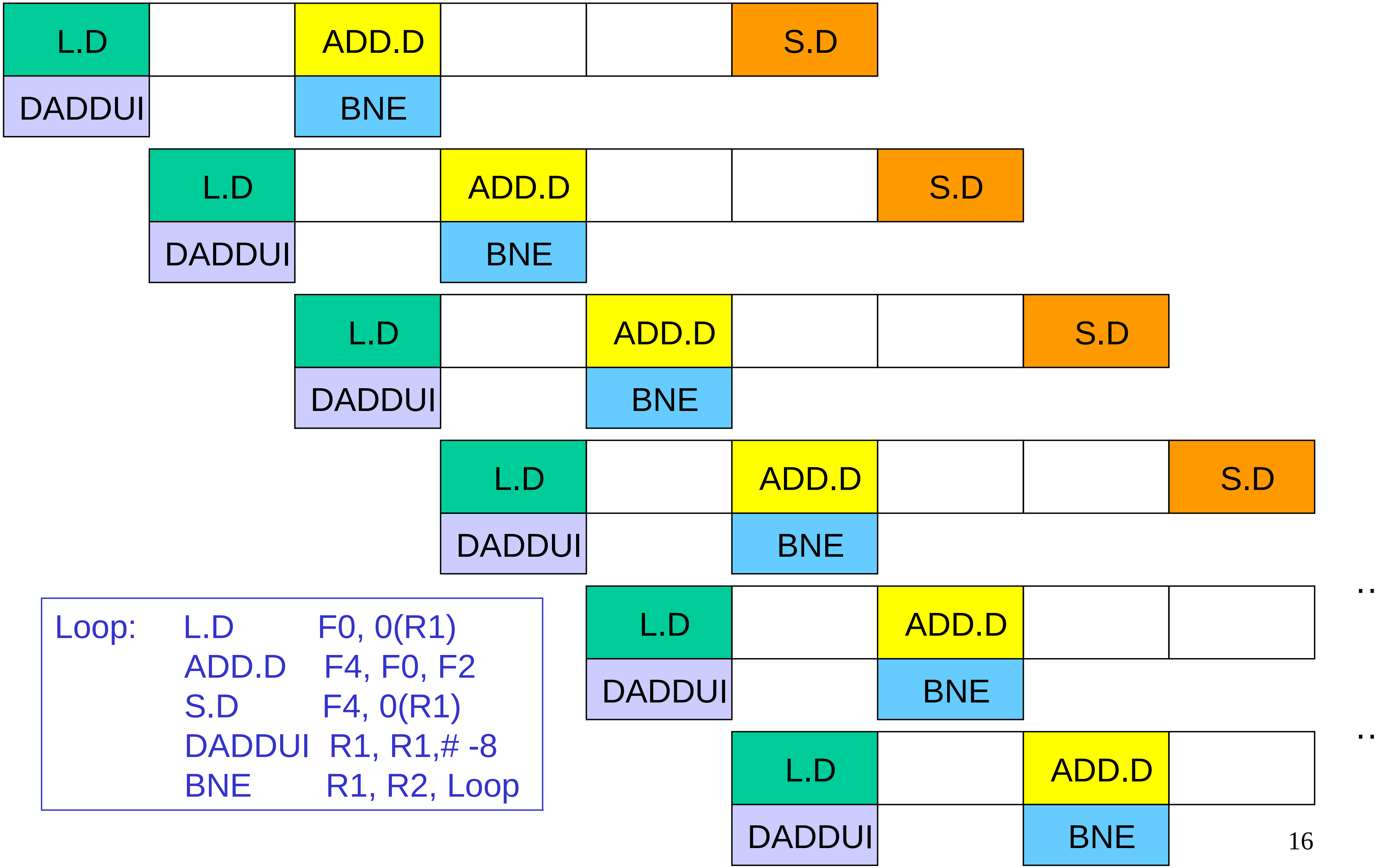
- What is the schedule with an unroll degree of 4?

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)		

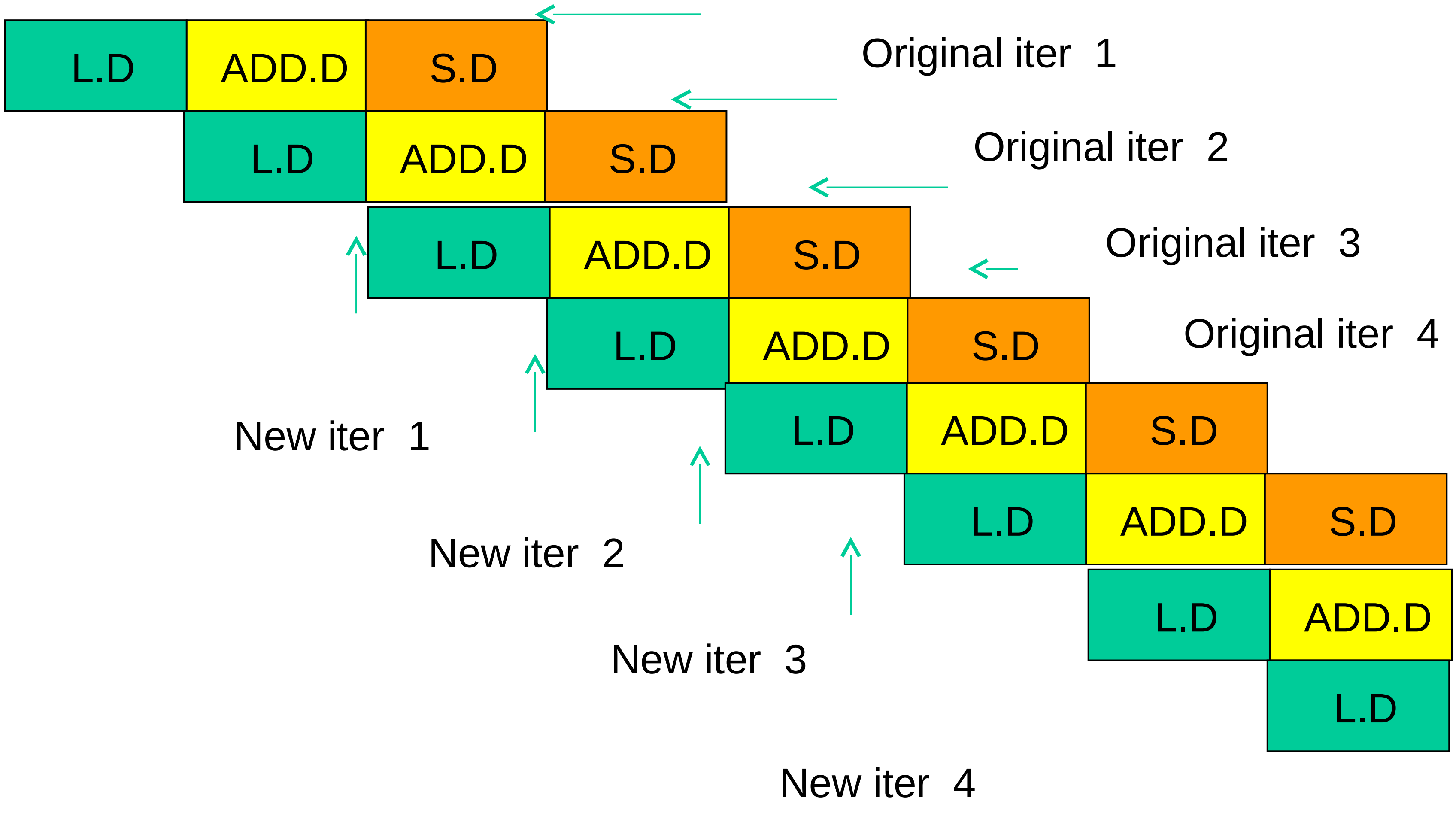
- Need unroll by degree 5 to eliminate stalls
- The compiler may specify instructions that can be issued as one packet
- The compiler may specify a fixed number of instructions in each packet: Very Large Instruction Word (VLIW)

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

Predication

- A branch within a loop can be problematic to schedule
- Control dependences are a problem because of the need to re-fetch on a mispredict
- For short loop bodies, control dependences can be converted to data dependences by using predicated/conditional instructions

Predicated or Conditional Instructions

```
if (R1 == 0)
  R2 = R2 + R4
else
  R6 = R3 + R5
  R4 = R2 + R3
```

→

```
R7 = !R1
R2 = R2 + R4   (predicated on R7)
R6 = R3 + R5   (predicated on R1)
R4 = R8 + R3   (predicated on R1)
```

Predicated or Conditional Instructions

- The instruction has an additional operand that determines whether the instr completes or gets converted into a no-op

- Example: lwc R1, 0(R2), R3   (load-word-conditional) will load the word at address (R2) into R1 if R3 is non-zero; if R3 is zero, the instruction becomes a no-op

- Replaces a control dependence with a data dependence (branches disappear) ; may need register copies for the condition or for values used by both directions

```
if (R1 == 0)
   R2 = R2 + R4
else
   R6 = R3 + R5
   R4 = R2 + R3
```

→

```
R7 = !R1 ;
R2 = R2 + R4   (predicated on R7)
R6 = R3 + R5   (predicated on R1)
R4 = R8 + R3   (predicated on R1)
```

Thank you!

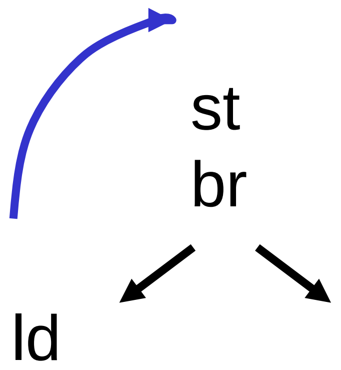
Complications

- Each instruction has one more input operand – more register ports/bypassing
- If the branch condition is not known, the instruction stalls (remember, these are in-order processors)
- Some implementations allow the instruction to continue without the branch condition and squash/complete later in the pipeline – wasted work
- Increases register pressure, activity on functional units
- Does not help if the br-condition takes a while to evaluate

Support for Speculation

- In general, when we re-order instructions, register renaming can ensure we do not violate register data dependences

- However, we need hardware support
  - to ensure that an exception is raised at the correct point
  - to ensure that we do not violate memory dependences



Detecting Exceptions

- Some exceptions require that the program be terminated (memory protection violation), while other exceptions require execution to resume (page faults)
- For a speculative instruction, in the latter case, servicing the exception only implies potential performance loss
- In the former case, you want to defer servicing the exception until you are sure the instruction is not speculative
- Note that a speculative instruction needs a special opcode to indicate that it is speculative

Program-Terminate Exceptions

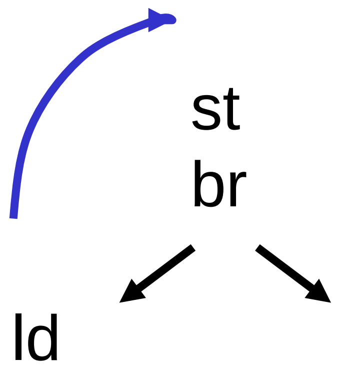
- When a speculative instruction experiences an exception, instead of servicing it, it writes a special NotAThing value (NAT) in the destination register
- If a non-speculative instruction reads a NAT, it flags the exception and the program terminates (it may not be desirable that the error is caused by an array access, but the segfault happens two procedures later)
- Alternatively, an instruction (the *sentinel*) in the speculative instruction's original location checks the register value and initiates recovery

Memory Dependence Detection
(Advanced Load Address Table)

In general, when we re-order instructions, register renaming can ensure we do not violate register data dependences

However, we need hardware support

- to ensure that an exception is raised at the correct point
- to ensure that we do not violate memory dependences



Memory Dependence Detection

- If a load is moved before a preceding store, we must ensure that the store writes to a non-conflicting address, else, the load has to re-execute

- When the speculative load issues, it stores its address in a table (Advanced Load Address Table in the IA-64)

- If a store finds its address in the ALAT, it indicates that a violation occurred for that address

- A special instruction (the *sentinel*) in the load's original location checks to see if the address had a violation and re-executes the load if necessary