Software Pipelining

Recap: Loop Unrolling with Scheduling

- Three different types of limits:
  - Decrease in the amount of overhead amortized with each unroll
    - If the loop is unrolled 8 times, the overhead is reduced from ½ cycles of the original iteration to ¼
  - The growth in code size due to loop unrolling (may increase cache miss rates)
  - Shortfall of registers created by aggressive unrolling and scheduling (register pressure)

Recap

- Loop unrolling improves the performance by eliminating overhead instructions
- Loop unrolling is a simple but useful method to increase the size of straight-line code fragments
- Sophisticated high-level transformations led to significant increase in complexity of the compilers

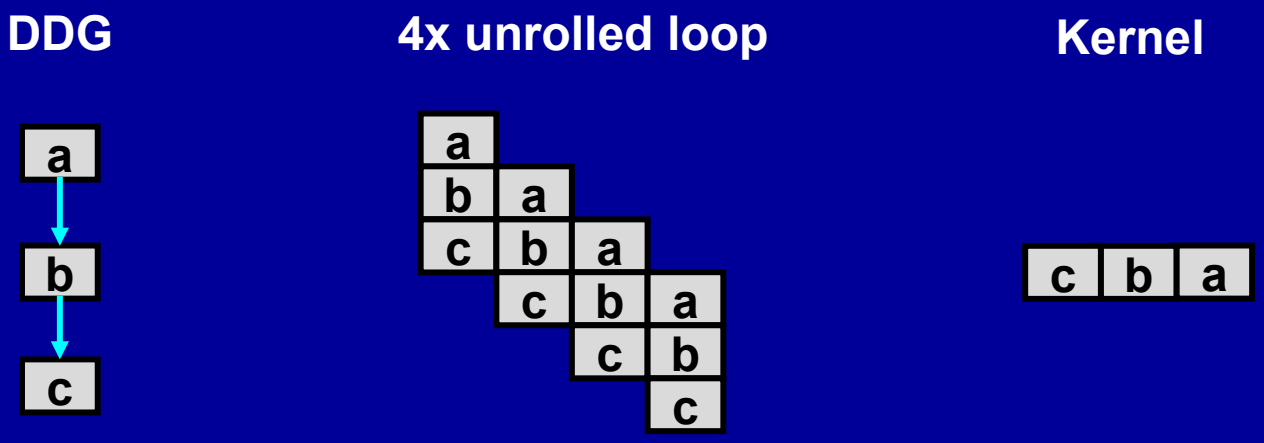
Software Pipelining

- Eliminates loop-independent dependence through code restructuring
  - Reduces stalls
  - Helps to achieve better performance in pipelined execution
  - As compared to simple loop unrolling:

  Consumes less code space

Software Pipelining

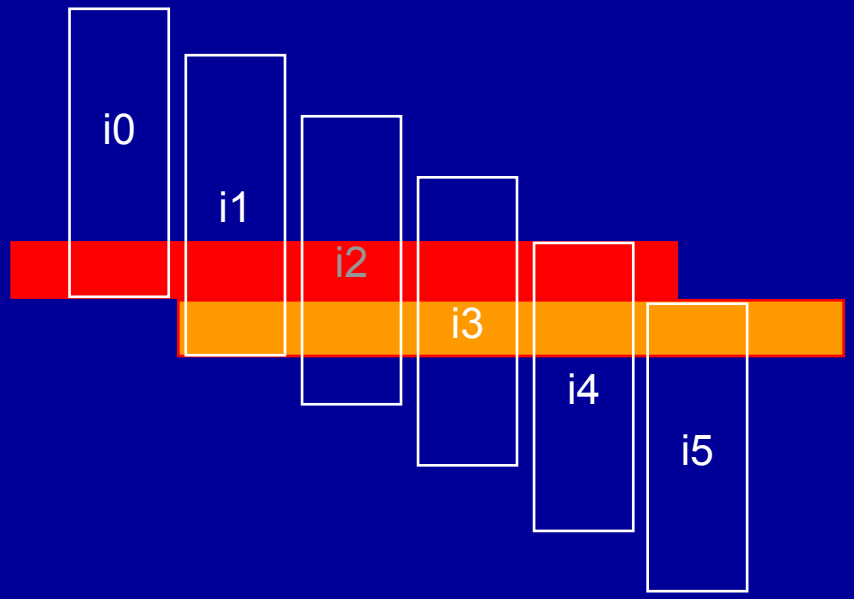
- Exactly just as it happens in a hardware pipeline:
  - In each iteration of a software pipelined code, some instruction of some iteration of the original loop is executed



Software Pipelining

- Central idea: reorganize loops
  - Each iteration is made from instructions chosen from different iterations of the original loop

Software Pipeline Iteration



Software Pipelining

- How is this done?

  1 → unroll loop body with an unroll factor of n. (we have taken n = 3 for our example)

  2 → select order of instructions from different iterations to pipeline

  3 → "paste" instructions from different iterations into the new pipelined loop body

Static Loop Unrolling Example

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

Software Pipelining: Step 1

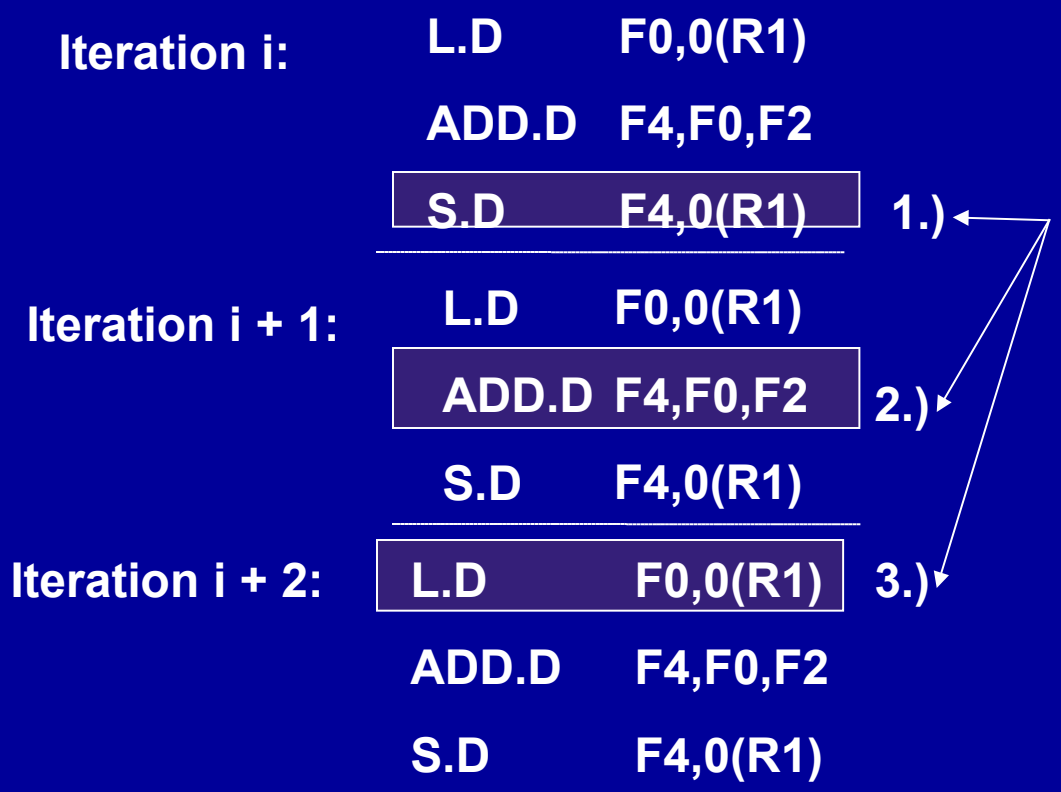
Iteration i:	L.D	F0,0(R1)
	ADD.D	F4,F0,F2
	S.D	F4,0(R1)
Iteration i + 1:	L.D	F0,0(R1)
	ADD.D	F4,F0,F2
	S.D	F4,0(R1)
Iteration i + 2:	L.D	F0,0(R1)
	ADD.D	F4,F0,F2
	S.D	F4,0(R1)

Note:

1. We are unrolling the loop. Hence no loop overhead Instructions are needed!

2. A single loop body of restructured loop would contain instructions from different iterations of the original loop body

Software Pipelining: Step 2

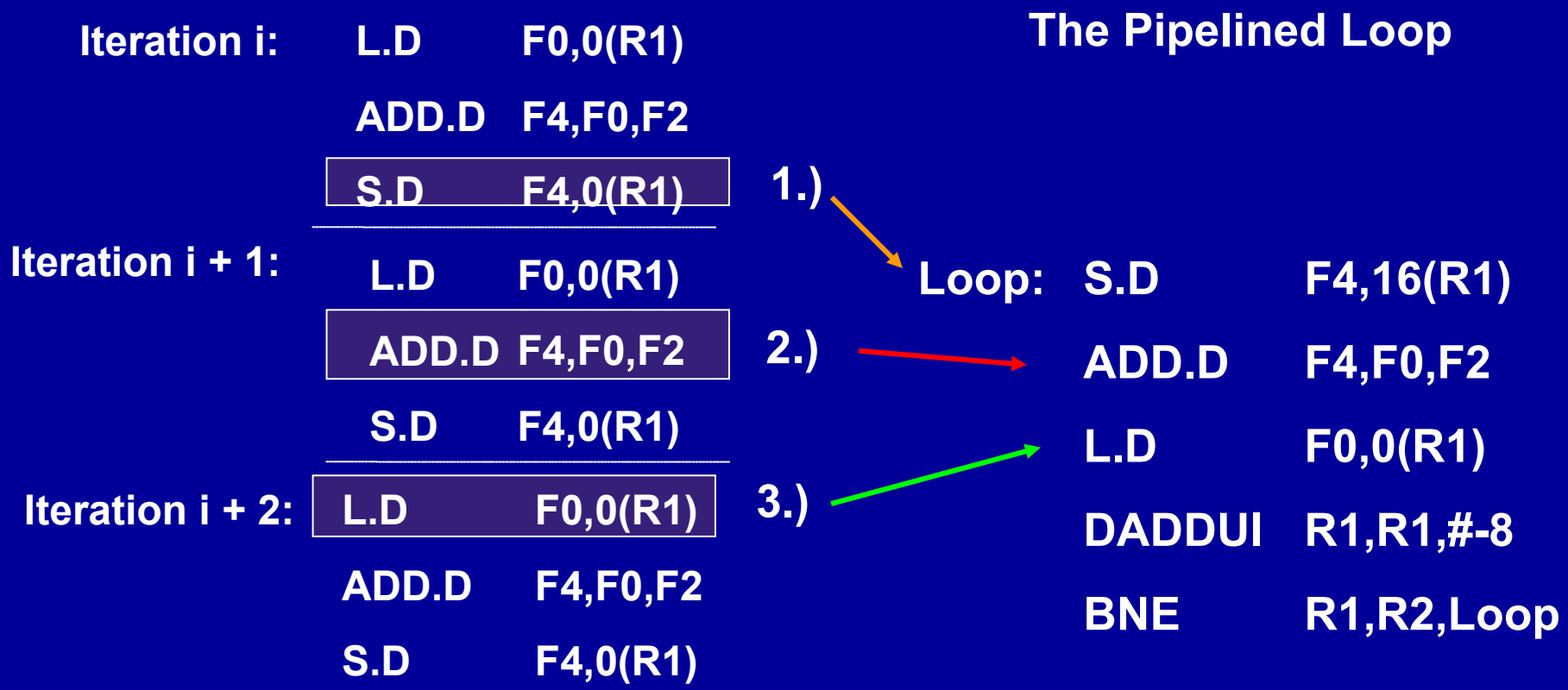


Notes:

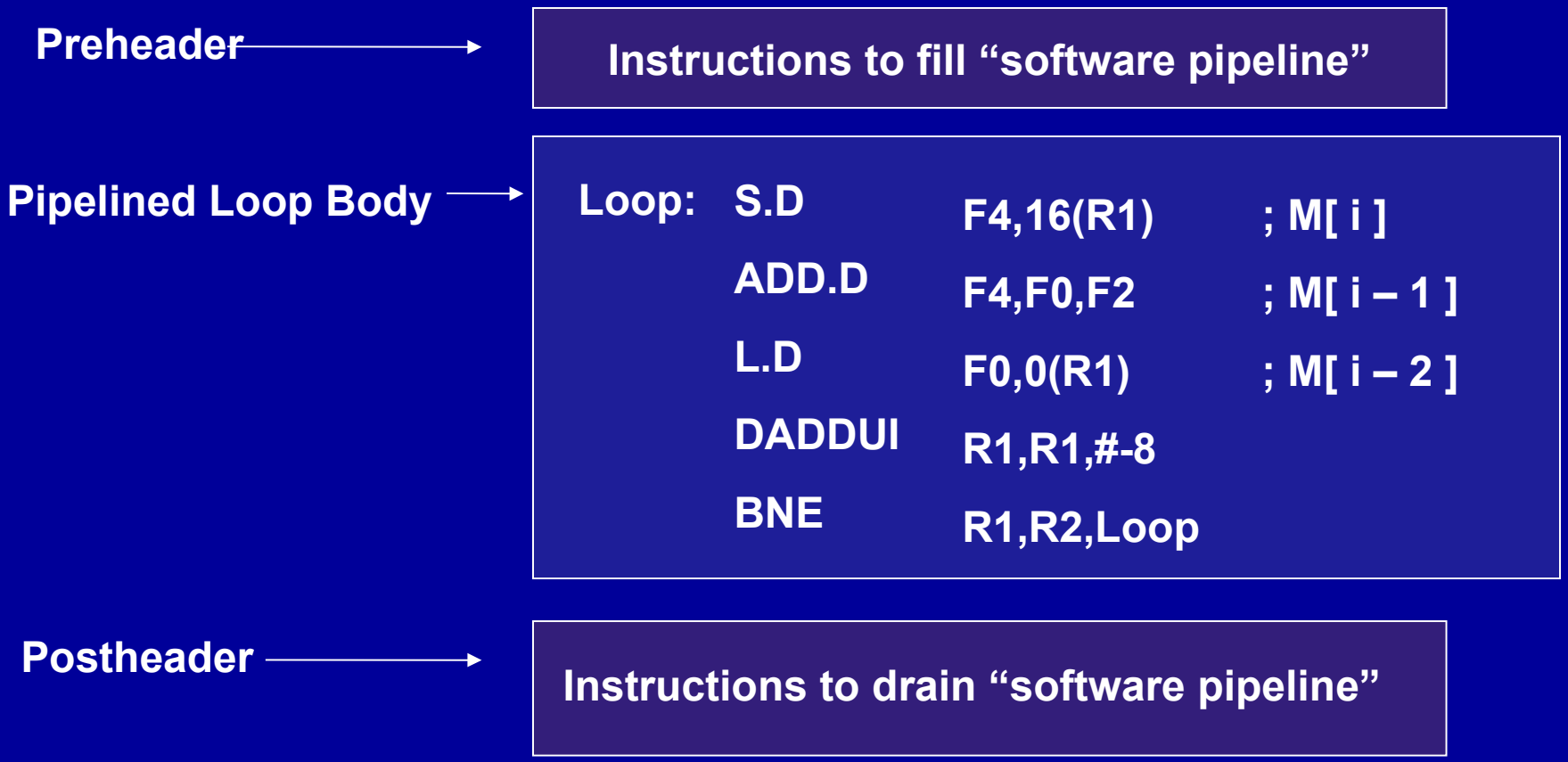
1. We'll select the following order in our pipelined loop:

2. Each instruction (L.D ADD.D S.D) must be selected at least once to make sure that we don't leave out any instruction of the original loop in the pipelined loop.

Software Pipelining: Step 3



Software Pipelining: Step 4



Software Pipelined Code

```
Loop :  S.D      F4,16(R1)     ; M[ i ]
        ADD.D    F4,F0,F2      ; M[ i – 1 ]
        L.D      F0,0(R1)      ; M[ i – 2 ]
        DADDUI   R1,R1,#-8
        BNE      R1,R2,Loop
```

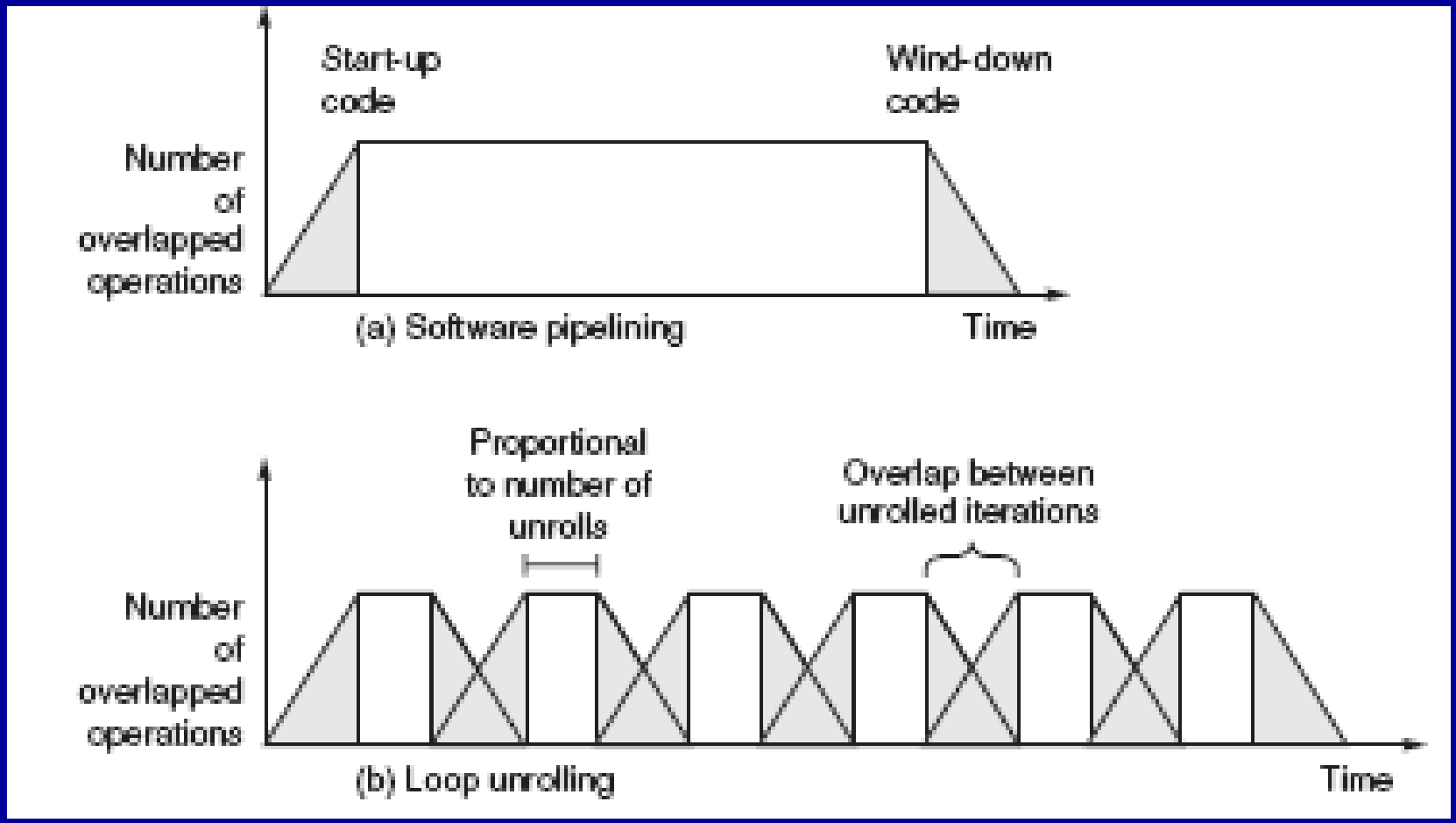
Software Pipelining Issues

- Register management can be tricky.
  - In more complex examples, we may need to increase the iterations between when data is read and when the results are used
- Optimal software pipelining has been shown to be an NP-complete problem:
  - Present solutions are based on heuristics

Software Pipelining Versus Loop Unrolling

- Software pipelining takes less code space.
- Software pipelining and loop unrolling reduce different types of inefficiencies:
  - Loop unrolling reduces loop management overheads
  - Software pipelining allows a pipeline to run at full efficiency by eliminating loop-independent dependencies

Software Pipelining Versus Loop Unrolling



Limitations of Scalar Pipelines

- Maximum throughput bounded by one instruction per cycle.
- Inefficient unification of instructions into one pipeline:
  - ALU, MEM stages very diverse e.g.: FP
- Rigid nature of in-order pipeline:
  - If a leading instruction is stalled, every subsequent instruction is stalled

Higher ILP Processor

- Pipelined Processors
  - An ideal CPI of 1 can be achieved by eliminating data stalls using the techniques discussed so far
- CPI less than one
  - To improve performance further we may try to achieve CPI less than 1
  - Two basic approaches:
    - Very Large Instruction Word (VLIW)
    - Superscalar

Two Paths to Higher ILP

- **VLIW:**
  - The compiler has complete responsibility of selecting a set of instructions to be executed concurrently
  - Simple hardware, smart compiler
- **Superscalar processors:**
  - Statically scheduled Superscalar processor
    - Multiple issue, in-order execution
  - Dynamically scheduled superscalar processor
    - Speculative execution, branch prediction
    - More hardware functionalities and complexities

Dynamic Instruction Scheduling: The Need

- We have seen that primitive pipelined processors tried to overcome data dependence through:
  - Forwarding:
    - But, many data dependences can not be overcome this way
  - Interlocking: brings down pipeline efficiency
- Software based instruction restructuring:
  - Handicapped due to inability to detect many dependences

Dynamic Instruction Scheduling

- **Scheduling:** Ordering the execution of instructions in a program so as to improve performance
- **Dynamic Scheduling:**
  - The hardware determines the order in which instructions execute
  - This is in contrast to statically scheduled processor where the compiler determines the order of execution

Points to Remember

- What is pipelining?
  - It is an implementation technique where multiple tasks are performed in an overlapped manner
- When can it be implemented?
  - It can be implemented when a task can be divided into two or subtasks, which can be performed independently
  - The earliest use of parallelism in designing CPUs (since 1985) to enhance processing speed was Pipelining
- Pipelining does not reduces the execution time of a single instruction, it increases the throughput
- CISC processors are not suitable for pipelining because of:
  - Variable instruction format
  - Variable execution time
  - Complex addressing mode
- RISC processors are suitable for pipelining because of:
  - Fixed instruction format
  - Fixed execution time
  - Limited addressing modes

Points to Remember

- There are situations, called hazards, that prevent the next instruction stream from getting executed in its designated clock cycle
- Three major types:
  - Structural hazards: Not enough HW resources to keep all instructions moving
  - Data hazards: Data results of earlier instructions not available yet
  - Control hazards: Control decisions resulting from earlier instruction (branches) not yet made; don't know which new instruction to execute
- Structural Hazard – can be overcome using additional hardware
- Data Hazards – can be overcome using additional hardware (forwarding) or software (compiler)

??