Instruction Level Parallelism

BITS Pilani
Pilani Campus

K Hari Babu
Department of Computer Science & Information Systems

Dynamic Scheduling

- A simple statically scheduled pipeline fetches an instruction and issues it, unless
  - there was a data dependence between an instruction already in the pipeline and the fetched instruction that cannot be hidden with bypassing or forwarding
  - If there is a data dependence that cannot be hidden, then the hazard detection hardware stalls the pipeline starting with the instruction that uses the result. No new instructions are fetched or issued until the dependence is cleared
- In dynamic scheduling, the hardware rearranges the instruction execution to reduce the stalls while maintaining data flow and exception behavior

Dynamic Scheduling

- Dynamic scheduling offers several advantages:
  - It enables handling some cases when dependences are unknown at compile time
    - for example, because they may involve a memory reference)
  - it allows the processor to tolerate unpredictable delays such as cache misses, by executing other code while waiting for the miss to resolve
  - allows code that was compiled with one pipeline in mind to run efficiently on a different pipeline
- Hardware speculation, a technique with significant performance advantages, is built on dynamic scheduling.
- The advantages of dynamic scheduling are gained at a cost of a significant increase in hardware complexity

Static vs Dynamic

- Dynamic scheduling takes runtime info into account
- Memory references are not resolved at compile time
  - LD R1 0(R2)
  - ST R3 0(R4)
  - Whether these instructions can be interchanged can't be known at compile time?
    - May be ST writing to same memory location before LD reads.
- Branches
  - Compiler can't predict branches accurately. Trace scheduling etc can be used, but lot of "undoing" work if mispredicted
- Latency due to cache misses
  - In dynamic scheduling, independent instructions can be scheduled
- Operations such as MUL with 0 or 1
  - Some operations happen in a fewer cycles compared to others

Dynamic Scheduling: The Idea

- A major limitation of simple pipelining techniques is that they use in-order instruction issue and execution:
  - Instructions are issued in program order, and if an instruction is stalled in the pipeline, no later instructions can proceed.
  - If there is a dependence between two closely spaced instructions in the pipeline, this will lead to a hazard and a stall will result
  - If there are multiple functional units, these units could lie idle
  - If instruction j depends on a long-running instruction i, currently in execution in the pipeline, then all instructions after j must be stalled until i is finished and j can execute.

Speculation

- Hardware-based speculation combines three key ideas:
  - dynamic branch prediction to choose which instructions to execute
  - speculation to allow the execution of instructions before the control dependences are resolved
    - with the ability to undo the effects of an incorrectly speculated sequence
  - dynamic scheduling to deal with the scheduling of different combinations of basic blocks.
- Dynamic scheduling without speculation vs with speculation
  - dynamic scheduling without speculation only partially overlaps basic blocks in the pipeline
    - because it requires that a branch be resolved before actually executing any instructions in the successor basic block
- Speculation is about crossing control dependencies and executing instructions

Speculation

```
1.       LD F4, 100(R4)    ;value not in cache…
2.       BLE F4, #0.66666, jump ;// branch if less or equal
3.       FADD F1, F1, #0.5
4.       DADD R1, R1, #2
5. jump: FADD F1, F1, #0.25
6.       DADD R1, R1, #1
```

- If there is cache miss at line 1, it might take 10s of cycles to fetch data from RAM
- One can use Branch Prediction on BLE, and start and complete the execution of either {3,4} or {5,6}
  - if this requires a number of clock cycles much shorter than that required to fetch the data for LD
- But what if the prediction is wrong?

innovate achieve lead

BITS Pilani
Pilani Campus

Multiple Issue (R3 3.7)

CPI<1

- Following can be used to eliminate stalls in a pipeline
  - Dynamic scheduling/Out of order execution
  - Speculative execution
- Pipeline throughput depends on slowest stage
  - Therefore some processors have very deep pipelines

- Long pipelines require accurate branch prediction for speculative filling.
- Misprediction leads to huge penalty.

Alternative: Use multiple pipelines

Microarchitecture	Pipeline stages
P5 (Pentium)	5
P6 (Pentium 3)	10
P6 (Pentium Pro)	14
NetBurst (Willamette)	20
NetBurst (Northwood)	20
NetBurst (Prescott)	31
NetBurst (Cedar Mill)	31
Core	14
Bonnell	16
Sandy Bridge	14
Silvermont	14 to 17
Haswell	14
Skylake	14
Kabylake	14

hardware - How long is a typical modern microprocessor pipeline? - Software Engineering Stack Exchange

CPI<1

- To improve performance further, we would like to decrease the CPI to less than one
  - But the CPI cannot be reduced below one if we issue only one instruction every clock cycle
  - This is possible if we duplicate some of the functional parts of the processor (e.g., have two ALUs), and have logic to issue several instructions concurrently

Multiple-issue Processors

- Multiple-issue processors come in three major flavors:
  - 1. statically scheduled superscalar processors
    - Limited to 2 instructions per clock cycle
  - 2. VLIW (very long instruction word) processors
  - 3. dynamically scheduled superscalar processors

- There are two general approaches to multiple issue:
  - static multiple issue (where the scheduling is done at compile time) and dynamic multiple issue (where the scheduling is done at execution time), also known as superscalar
- Intel Core 2 processors are superscalar and can issue up to 4 instructions per clock cycle i.e . 4-way superscalar.

Multiple-issue Processors

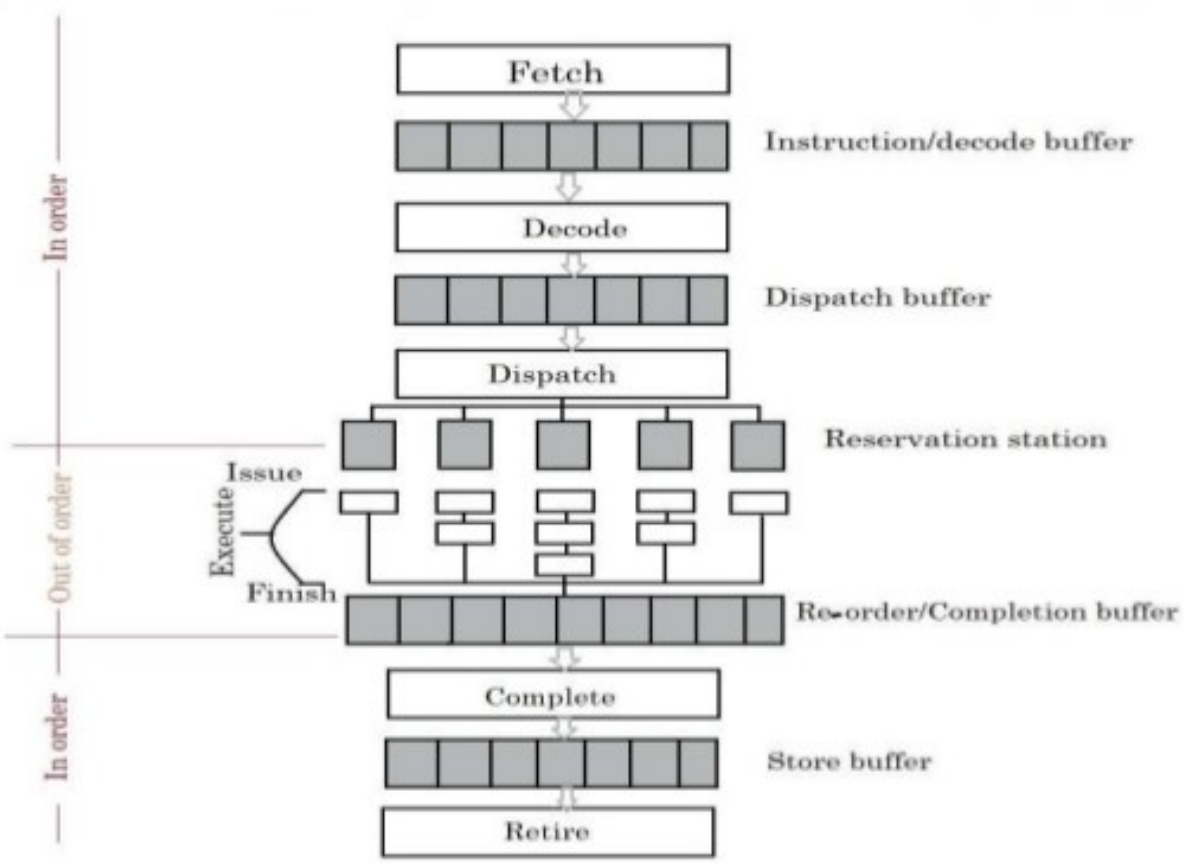
- Multiple-issue processors come in three major flavors:
  - 1. statically scheduled superscalar processors
  - 2. VLIW (very long instruction word) processors
  - 3. dynamically scheduled superscalar processors
- (1) and (3) issue varying numbers of instructions per clock
  - (1) uses in-order execution as given by compiler
  - (3) uses out-of- order execution at run time
- VLIW processors, in contrast, issue a fixed number of instructions formatted either as one large instruction or as a fixed instruction packet with the parallelism among instructions explicitly indicated by the instruction
  - VLIW processors are inherently statically scheduled by the compiler
  - When Intel and HP created the IA-64 architecture EPIC—explicitly parallel instruction computer—for this architectural style

Super-scalar Processor

BITS Pilani

- Superscalar is a term coined in the late 1980s
  - All current desktop and server market processors are now superscalar.

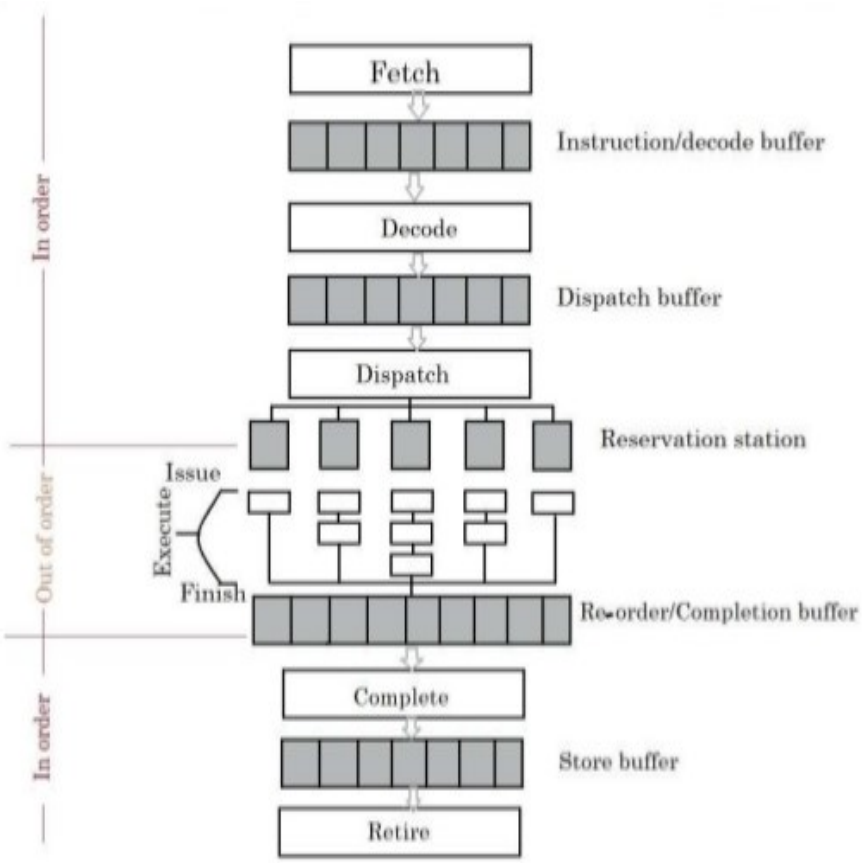
By taking advantage of available memory bandwidth, fetch a number of instructions per clock cycle, and start executing them



Super-scalar Processor

- Requires multiple arithmetic units and register files with additional ports to avoid structural hazards

- issue instructions (in order) to reservation / functional units
- execute instructions out of order as operands are available
- commit results (in order) to registers / memory
- feature of all current x86 architectures



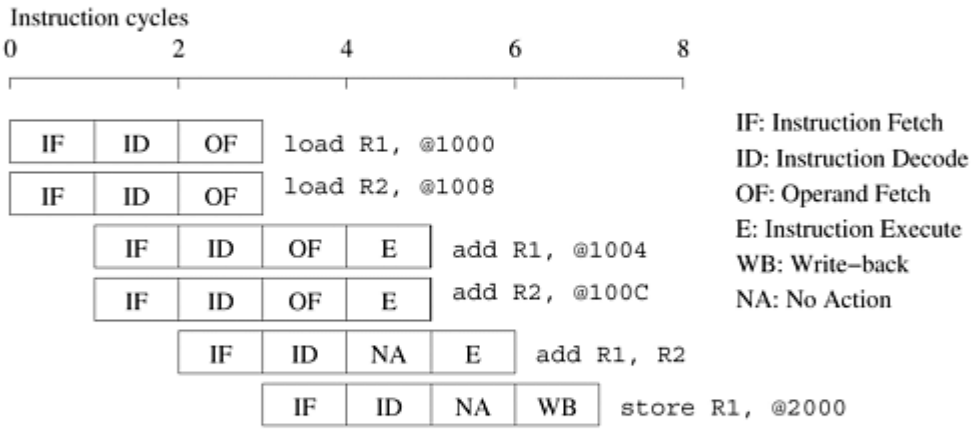
Example

- Consider a processor with two pipelines and the ability to simultaneously issue two instructions. Since the architecture allows two issues per clock cycle, it is also referred to as two-way superscalar or dual issue execution.

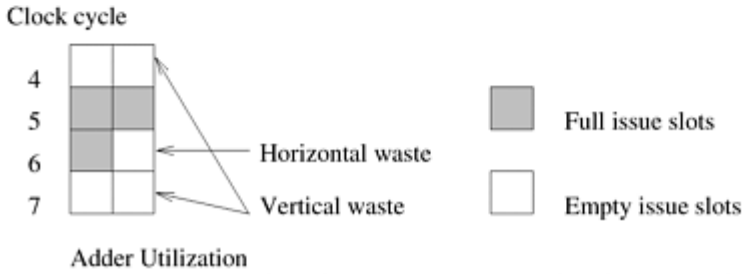
```
1. load R1, @1000        1. load R1, @1000        1. load R1, @1000
2. load R2, @1008        2. add R1, @1004         2. add R1, @1004
3. add R1, @1004         3. add R1, @1008         3. load R2, @1008
4. add R2, @100C         4. add R1, @100C         4. add R2, @100C
5. add R1, R2            5. store R1, @2000       5. add R1, R2
6. store R1, @2000                                6. store R1, @2000
```

(i) (ii) (iii)

(a) Three different code fragments for adding a list of four numbers.



(b) Execution schedule for code fragment (i) above.



(c) Hardware utilization trace for schedule in (b).

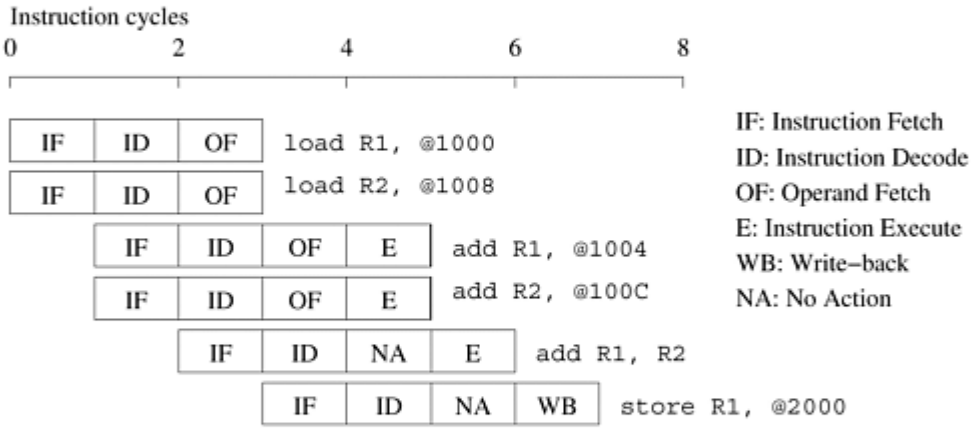
Example

- Consider the execution of the first code fragment in for adding four numbers. The first and second instructions are independent and therefore can be issued concurrently.
  - simultaneous issue of the instructions load R1, @1000 and load R2, @1008 at t = 0. The instructions are fetched, decoded, and the operands are fetched.
- The next two instructions, add R1, @1004 and add R2, @100C are also mutually independent, although they must be executed after the first two instructions. Consequently, they can be issued concurrently at t = 1 since the processors are pipelined.

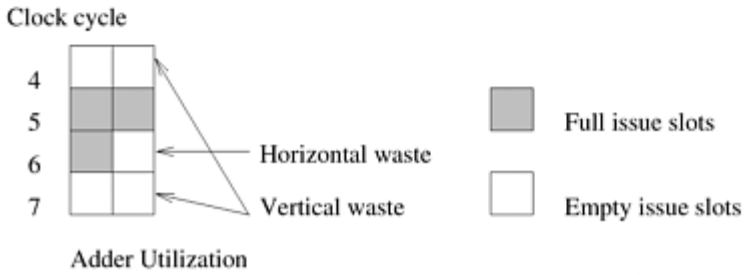
```
1. load R1, @1000      1. load R1, @1000      1. load R1, @1000
2. load R2, @1008      2. add R1, @1004       2. add R1, @1004
3. add R1, @1004       3. add R1, @1008       3. load R2, @1008
4. add R2, @100C       4. add R1, @100C       4. add R2, @100C
5. add R1, R2          5. store R1, @2000     5. add R1, R2
6. store R1, @2000                            6. store R1, @2000
```

(i) (ii) (iii)

(a) Three different code fragments for adding a list of four numbers.



(b) Execution schedule for code fragment (i) above.



(c) Hardware utilization trace for schedule in (b).

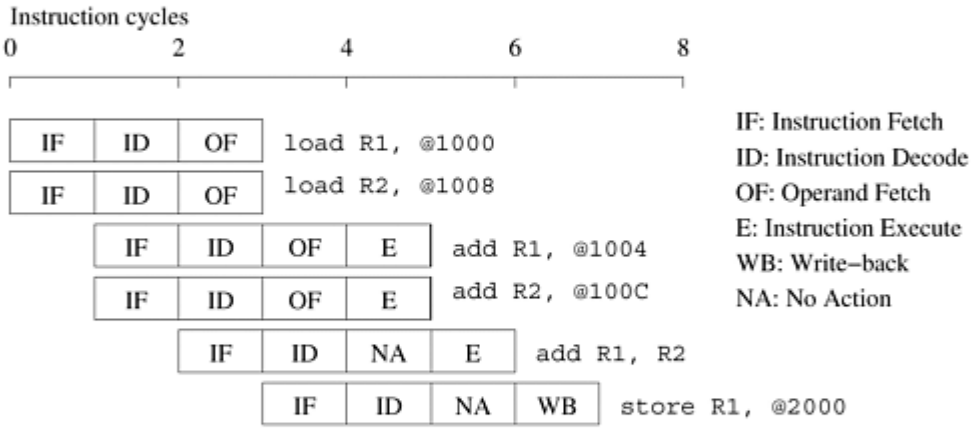
Example

- The next two instructions, add R1, R2 and store R1, @2000 cannot be executed concurrently
  - Therefore, only the add instruction is issued at t = 2 and the store instruction at t = 3. Note that the instruction add R1, R2 can be executed only after the previous two instructions have been executed.
- The schedule assumes that each memory access takes a single cycle. In reality, this may not be the case. the ability to simultaneously issue two instructions.

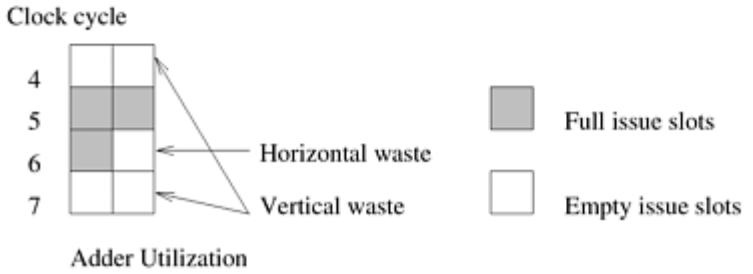
```
1. load R1, @1000      1. load R1, @1000      1. load R1, @1000
2. load R2, @1008      2. add R1, @1004       2. add R1, @1004
3. add R1, @1004       3. add R1, @1008       3. load R2, @1008
4. add R2, @100C       4. add R1, @100C       4. add R2, @100C
5. add R1, R2          5. store R1, @2000     5. add R1, R2
6. store R1, @2000                            6. store R1, @2000
```

(i) (ii) (iii)

(a) Three different code fragments for adding a list of four numbers.



(b) Execution schedule for code fragment (i) above.



(c) Hardware utilization trace for schedule in (b).

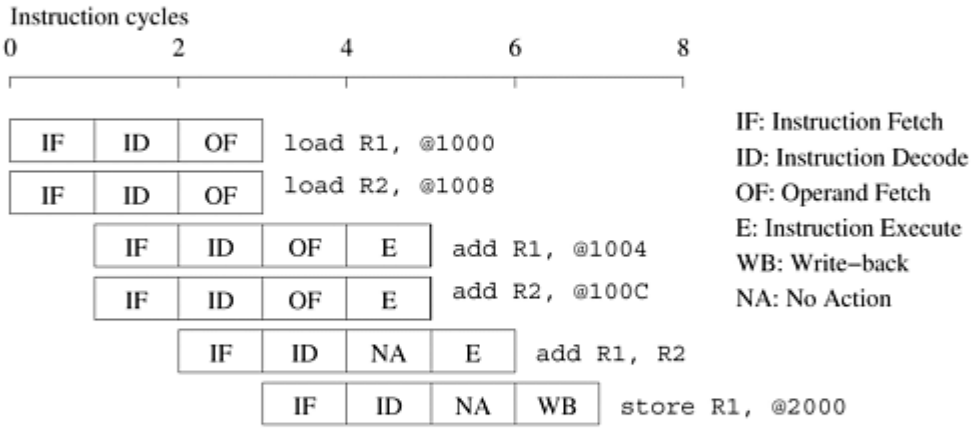
Example

- What about second code fragment?
- Are there any parallel instructions?
- The amount of instruction level parallelism in a program is often limited and is a function of coding technique. In the second code fragment, there can be no simultaneous issue, leading to poor resource utilization.

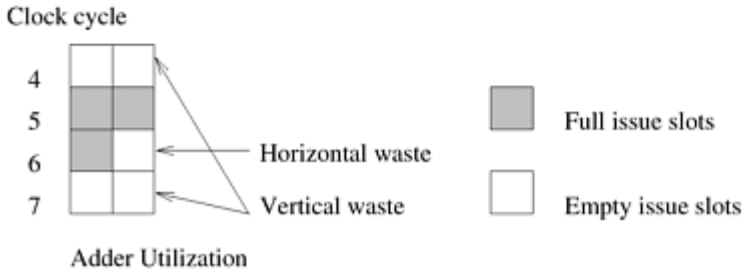
```
1. load R1, @1000      1. load R1, @1000      1. load R1, @1000
2. load R2, @1008      2. add R1, @1004       2. add R1, @1004
3. add R1, @1004       3. add R1, @1008       3. load R2, @1008
4. add R2, @100C       4. add R1, @100C       4. add R2, @100C
5. add R1, R2          5. store R1, @2000     5. add R1, R2
6. store R1, @2000                            6. store R1, @2000
```

(i) (ii) (iii)

(a) Three different code fragments for adding a list of four numbers.



(b) Execution schedule for code fragment (i) above.



(c) Hardware utilization trace for schedule in (b).

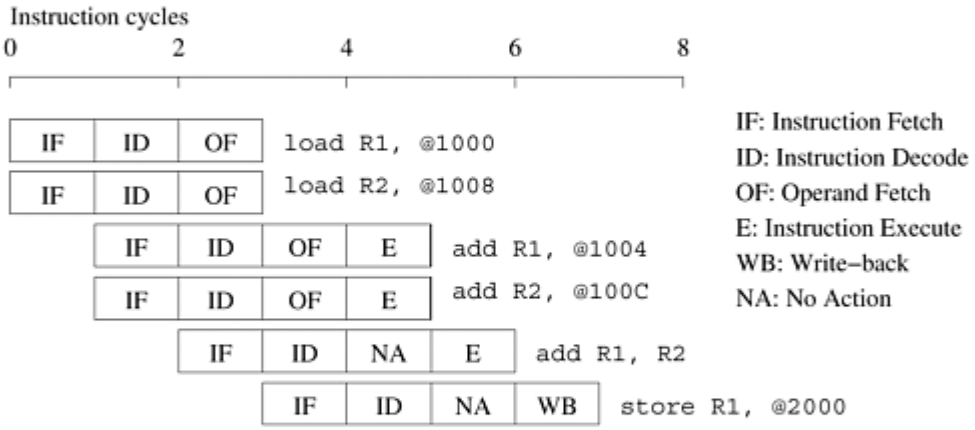
Example

- What about second code fragment?
- Are there any parallel instructions?
- The amount of instruction level parallelism in a program is often limited and is a function of coding technique. In the second code fragment, there can be no simultaneous issue, leading to poor resource utilization.

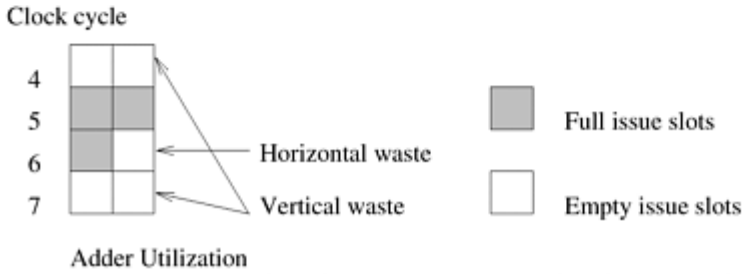
```
1. load R1, @1000      1. load R1, @1000      1. load R1, @1000
2. load R2, @1008      2. add R1, @1004       2. add R1, @1004
3. add R1, @1004       3. add R1, @1008       3. load R2, @1008
4. add R2, @100C       4. add R1, @100C       4. add R2, @100C
5. add R1, R2          5. store R1, @2000     5. add R1, R2
6. store R1, @2000                            6. store R1, @2000
```

(i) (ii) (iii)

(a) Three different code fragments for adding a list of four numbers.



(b) Execution schedule for code fragment (i) above.



(c) Hardware utilization trace for schedule in (b).

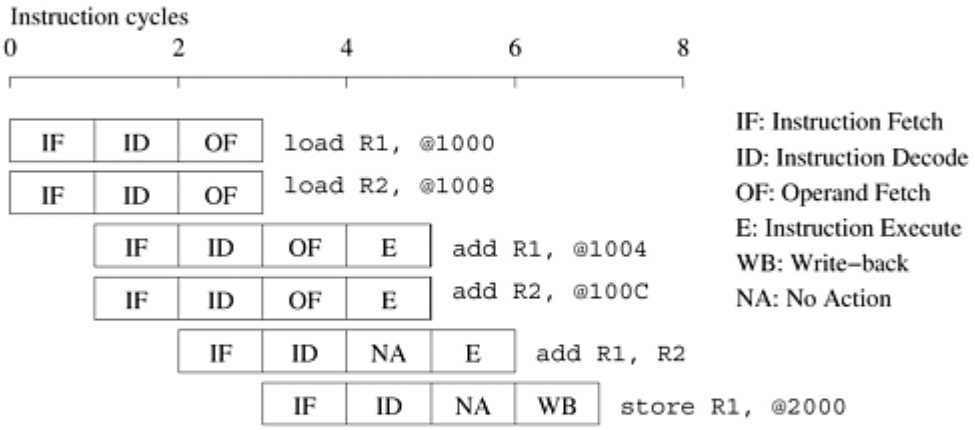
Example

- Third code fragment
- there is a data dependency between the first two instructions - load R1, @1000 and add R1, @1004. Therefore, these instructions cannot be issued simultaneously.
- The processor needs the ability to issue instructions ***out-of-order*** to accomplish desired reordering. The parallelism available in ***in-order*** issue of instructions can be highly limited.
  - Same ordering as first fragment can be obtained
- Most current microprocessors are capable of out-of-order issue and completion

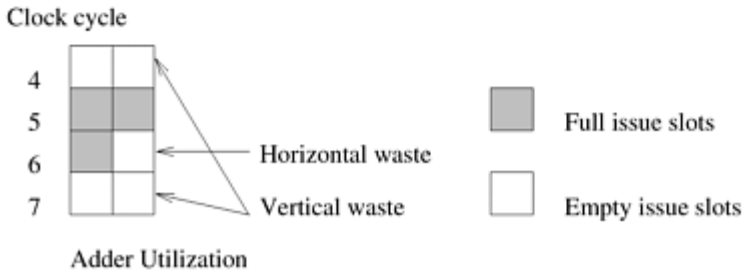
```
1. load R1, @1000      1. load R1, @1000      1. load R1, @1000
2. load R2, @1008      2. add R1, @1004       2. add R1, @1004
3. add R1, @1004       3. add R1, @1008       3. load R2, @1008
4. add R2, @100C       4. add R1, @100C       4. add R2, @100C
5. add R1, R2          5. store R1, @2000     5. add R1, R2
6. store R1, @2000                            6. store R1, @2000
```

(i) (ii) (iii)

(a) Three different code fragments for adding a list of four numbers.



(b) Execution schedule for code fragment (i) above.



(c) Hardware utilization trace for schedule in (b).

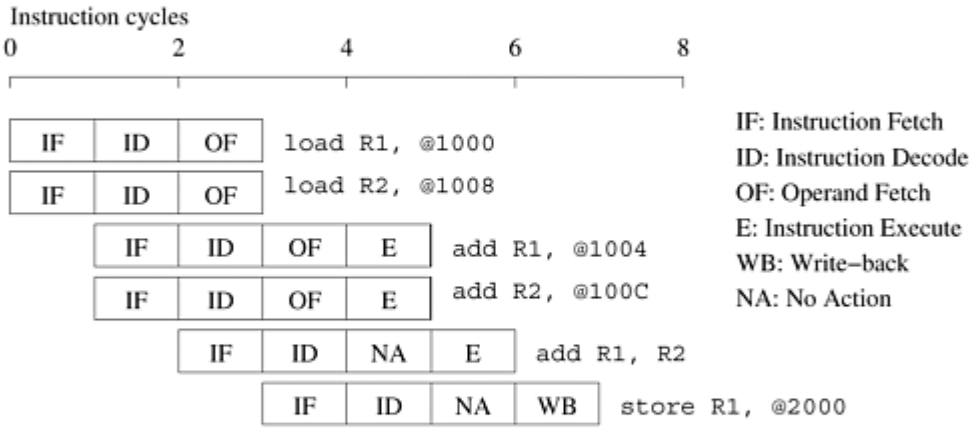
Example

- Two ALU units:
  - If, during a particular cycle, no instructions are issued on the execution units →vertical waste
  - If only part of the execution units are used during a cycle, →horizontal waste.
  - In the example, we have two cycles of vertical waste and one cycle with horizontal waste.
  - In all, only three of the eight available cycles are used for computation. This implies that the code fragment will yield no more than three-eighths of the peak rated FLOP count of the processor.
  - Due to limited parallelism, resource dependencies, or the inability of a processor to extract parallelism, the resources of superscalar processors are heavily under-utilized

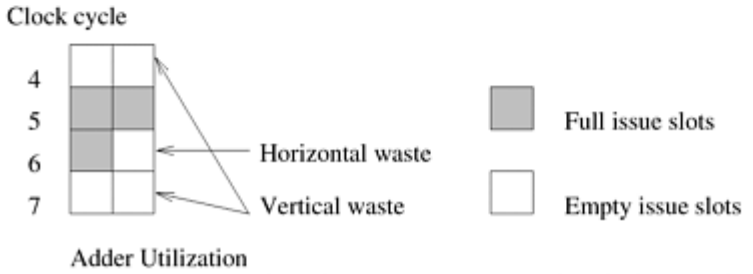
```
1. load R1, @1000        1. load R1, @1000        1. load R1, @1000
2. load R2, @1008        2. add R1, @1004         2. add R1, @1004
3. add R1, @1004         3. add R1, @1008         3. load R2, @1008
4. add R2, @100C         4. add R1, @100C         4. add R2, @100C
5. add R1, R2            5. store R1, @2000       5. add R1, R2
6. store R1, @2000                                6. store R1, @2000
```

(i) (ii) (iii)

(a) Three different code fragments for adding a list of four numbers.



(b) Execution schedule for code fragment (i) above.



(c) Hardware utilization trace for schedule in (b).

VLIW Processors

- The parallelism extracted by superscalar processors
  - limited by the instruction look-ahead window
  - hardware logic for dynamic dependency analysis is of 5-10% of the total logic on conventional microprocessors (about 5% on the four-way superscalar Sun UltraSPARC). This complexity grows roughly quadratically with the number of issues and can become a bottleneck.
- An alternate concept for exploring instruction-level parallelism used in very long instruction word (VLIW) processors relies on the compiler to resolve dependencies and resource availability at compile time.
  - Since scheduling is done in software, the decoding and instruction issue mechanisms are simpler in VLIW processors. The compiler has a larger context from which to select instructions and can use a variety of transformations to optimize parallelism when compared to a hardware issue unit

VLIW Processors

- Compilers do not have the dynamic program state (e.g., the branch history buffer) available to make scheduling decisions. This reduces the accuracy of branch and memory prediction, but allows the use of more sophisticated static prediction schemes.
  - Other runtime situations such as stalls on data fetch because of cache misses are extremely difficult to predict accurately. This limits the scope and performance of static compiler-based scheduling.
- Loop unrolling, branch prediction and speculative execution all play important roles in the performance of VLIW processors.
- Superscalar and VLIW processors have been successful in exploiting implicit parallelism.
  - They are generally limited to smaller scales of concurrency in the range of four- to eight-way parallelism.

Q&A

innovate achieve lead

BITS Pilani
Pilani Campus

Thank You