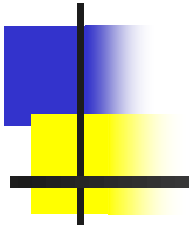


6 : The Processor (Advanced)

Rajesh Panicker,
NUS

CG3207

Note :

- Slides not covered in detail in the class are left as a self-learning exercise

Acknowledgement :

- Text by Harris and Harris and companion materials
- Text by Patterson & Hennessy and companion materials by Mary Jane Irwin, PSU

How to Extract **Even More** Performance?

- Branch Prediction
- Deep Pipelining
- Micro-operations
- Superscalar Processor
  - Out of Order Processor
  - Register Renaming
- VLIW Processor
- Hardware Multithreading
- Multicore / processor Systems (Chapter 7)

Multiple Issue Processors

Control Hazards : Static Branch Prediction

- Resolve branch hazards by assuming a given outcome and proceeding without waiting to see the actual branch outcome

1. Predict not taken – always predict branches will not be taken, continue to fetch from the sequential instruction stream, only when branch *is* taken does the pipeline stall
   - If taken, flush instructions after the branch (earlier stage of execution in the pipeline)
   - Ensure that those flushed instructions haven't changed the machine state – stall if required (for example, instructions which write to flags in E, `STR` etc).
   - Restart the pipeline at the branch destination
   - We had implicitly implemented this scheme in the earlier design!
2. Predict taken – predict branches will always be taken
   - *Where* will the branch be taken to? – need some mechanism to predict that too!

Predict Not Taken

- "Predict not taken" works well for "top of the loop" branching structures
  - `for` loop, `while` loop

```
Loop:  CMP R1, R2
       BEQ Out
       1st loop instr
            .
            .
            .
       last loop instr
       B   Loop
Out:   fall out instr
```

- "Predict not taken" doesn't work well for "bottom of the loop" branching structures
  - `do-while` loop

```
Loop:  1st loop instr
       2nd loop instr
            .
            .
            .
       last loop instr
       CMP  R1, R2
       BNE  Loop
       fall out instr
```

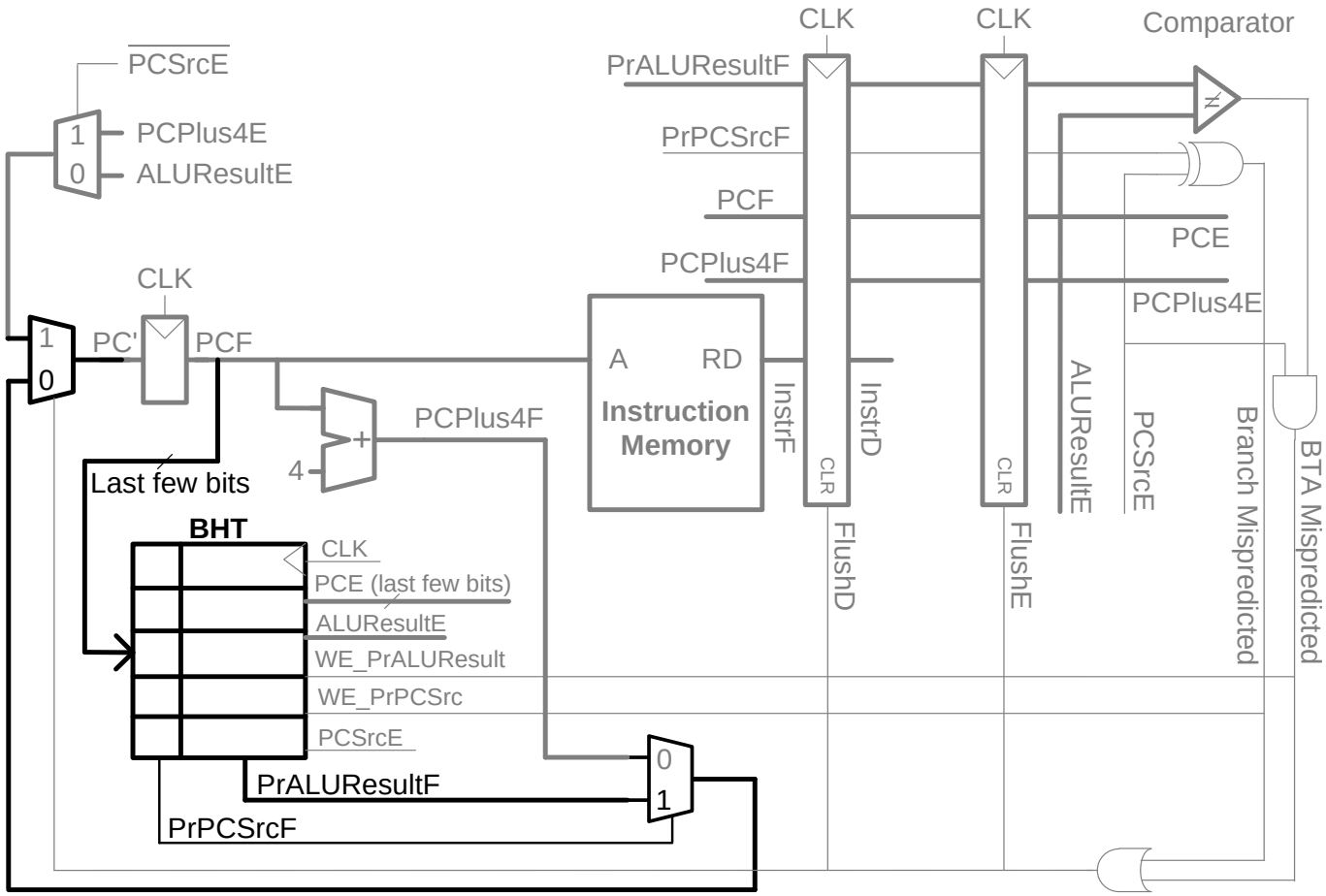
Dynamic Branch Prediction

- As the branch penalty increases (i.e., for deeper pipelines), a simple static prediction scheme will hurt performance. With more hardware, it is possible to try to predict branch behavior dynamically during program execution (i.e., using *run-time* information)
- A branch prediction buffer (aka branch history table (BHT)) in the F stage addressed/indexed by a few lower bits of the PC, contains bit(s) that tells *whether* the branch was taken the last time it was executed
- Similarly, a branch target buffer (BTB) caches the BTA computed when the branch was taken last time. It gives a prediction of *where* the branch is taken to. It is often incorporated into the BHT itself
- If the prediction is perfect, stalls can be avoided entirely

When Prediction Goes Wrong

- Branch / BTA prediction could be wrong
  - May be a wrong prediction for this branch this iteration
  - May be from a different instruction (possibly not even branch) with the same lower order PC bits
  - In any case, we cannot compromise on *correctness*, though we can compromise on *performance*
- If the prediction is wrong
  - Flush the incorrect instruction(s) in pipeline
  - Restart the pipeline with the correct PC value
  - Update the prediction bit(s) and / or predicted branch target
- A 4096 entry (12-bit) BHT varies from 1% misprediction (nasa7, tomcatv) to 18% (eqntott)
  - 4096 entry table is about as good as an infinite table, but 4096 entries would need a lot of hardware. Also, bigger tables -> slower

Branch History Table



PrALUResultF = Predicted ALUResult = Predicted Branch Target Address (BTA)
PrPCSrcF = Predicted PCSrc = Branch Prediction

1-bit Prediction Accuracy

- A 1-bit predictor will be incorrect twice
  - Assume predict_bit = 0 to start (indicating branch not taken) and loop control is at the bottom of the loop code
  1. First time through the loop, the predictor mispredicts the branch since the branch is taken back to the top of the loop; invert prediction bit (predict_bit = 1)
  2. As long as branch is taken (looping), prediction is correct
  3. Exiting the loop, the predictor again mispredicts the branch since this time the branch is not taken falling out of the loop; invert prediction bit (predict_bit = 0)

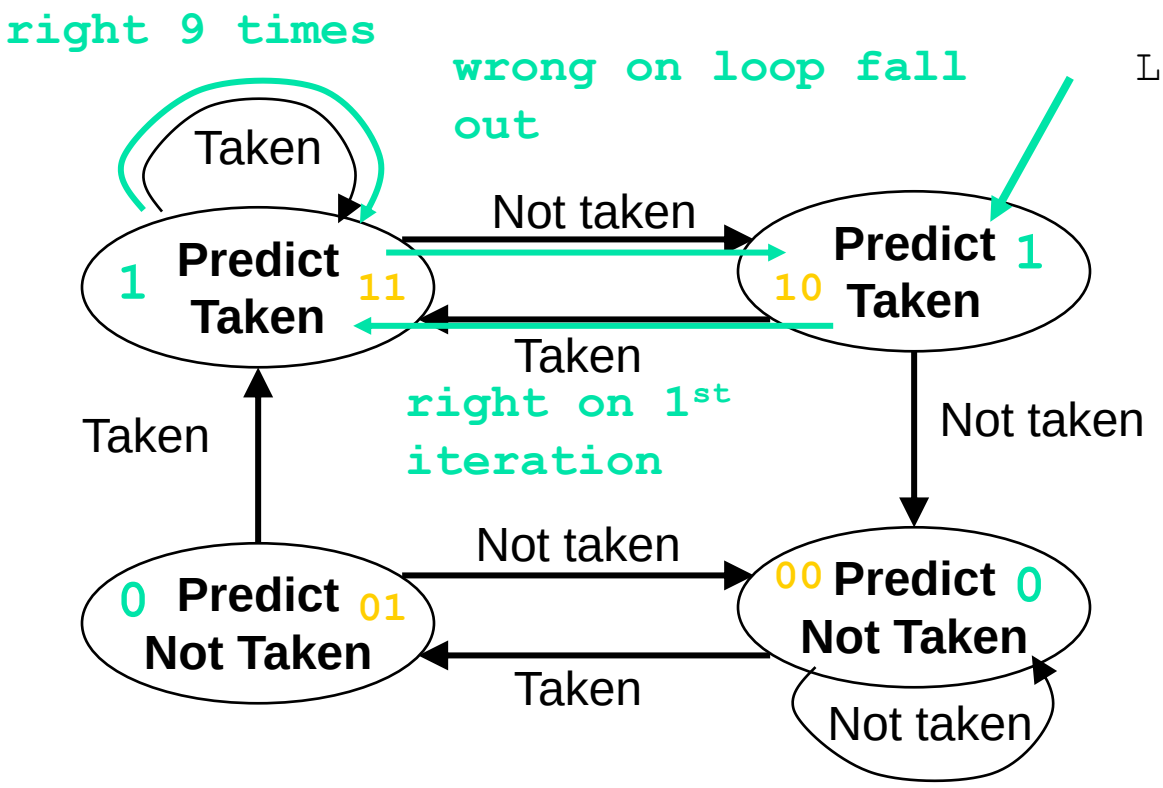
```
        MOV  R1, #10
Loop:   1st loop instr
        2nd loop instr
             .
             .
             .
        last loop instr
        SUBS R1, #1
        BNE  Loop
        fall out instr
```

Prediction correct?
First : NYYYYYYYYN
Subsequent : NYYYYYYYYN

- For 10 times through the loop we have a 80% prediction accuracy for a branch that is taken 90% of the time

2-bit Predictors

- A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed




```
        MOV  R1, #10
Loop:   1st loop instr
        2nd loop instr
              .
              .
              .
        last loop instr
        SUBS R1, #1
        BNE  Loop
        fall out instr
```

Prediction correct?

First : NNYYYYYYYN
(starting state : 00)

Subsequent : YYYYYYYYYN
(starting state : 10)

- BHT stores the FSM state
- This approach is usually called a saturating counter

Deep Pipelining

- 10-20 stages typical. Diminishing returns with additional stages
  - Sequencing overhead – Pipeline register setup time and (clock to Q) propagation delay becomes significant
    - The share of actual logic in the clock period decreases – increased latency
  - Pipeline hazards – increased possibility of data hazard, requiring more hazard and forwarding logic
    - Delay caused by forwarding logic (such as muxes) becomes significant too, diminishing returns further
    - Increased power and cost from hazard/forwarding logic
    - Higher misprediction penalty – need to flush more instructions -> better branch prediction required -> even more hardware
  - Example : Intel Pentium 4 (Netburst)
    - Upto 31 pipeline stages, 20 cycles misprediction penalty

Micro-operations

- Decompose more complex instructions into a series of simple instructions called *micro-operations* (*micro-ops* or *µ-ops*)
- At run-time, complex instructions are decoded into one or more micro-ops
- Used heavily in CISC (complex instruction set computer) architectures (e.g., x86)
- Used for some ARM instructions, for example:

Complex Op	Micro-op Sequence
`LDR R1, [R2], #4`	`LDR R1, [R2]` `ADD R2, R2, #4`

Without u-ops, would need 2nd write port on the register file

Micro-operations …

- Decoding done by hardware - the programmer/compiler doesn't need to know what is under the hood. The micro-operations need not even be valid instructions in the ISA
- Allow for dense code (fewer memory accesses)
- Yet preserve simplicity of RISC hardware
- Modern x86 processors include a micro-op cache for decoded micro-ops
  - An instruction present in the cache need not be decoded again!

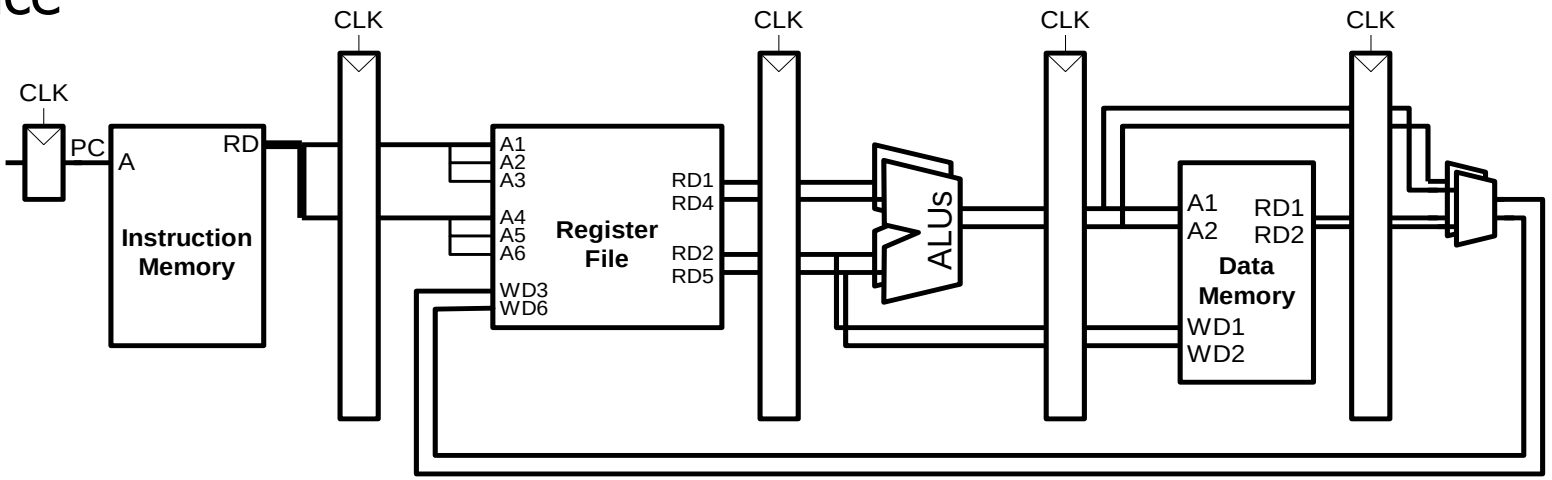
Micro-operations vs pseudo-instructions

The assembler splits pseudo-instructions (which are **not valid instructions** in the ISA) **into valid instructions** within the ISA. The instructions to which the pseudo-instruction is split into is visible to the programmer.

In case of micro-operations, the hardware splits the complex instructions (which are **valid instructions** in the ISA) into simple instructions/operations which are **not necessarily within the ISA**. Micro-operations are invisible to the programmer.

Multiple-Issue Processors

- Fetch (and execute) more than one instructions at one time (expand every pipeline stage to accommodate multiple instructions) – multiple-issue
- Multiple copies of datapath execute multiple instructions at once
  - The instruction execution rate, CPI, will be less than 1, so instead we use IPC: instructions per clock cycle
- Dependencies make it tricky to issue multiple instructions at once



Types of Parallelism

- Instruction-level parallelism (ILP) of a program – a measure of the average number of instructions in a program that a processor *might* be able to execute at the same time
  - Mostly determined by the number of true (data) dependencies and procedural (control) dependencies in relation to the number of other instructions. Usually < 3
- Machine parallelism of a processor – a measure of the ability of the processor to take advantage of the ILP of the program
  - Determined by the number of instructions that can be fetched and executed at the same time
- To achieve high performance, need *both* ILP and machine parallelism

Multiple-Issue Processor Styles

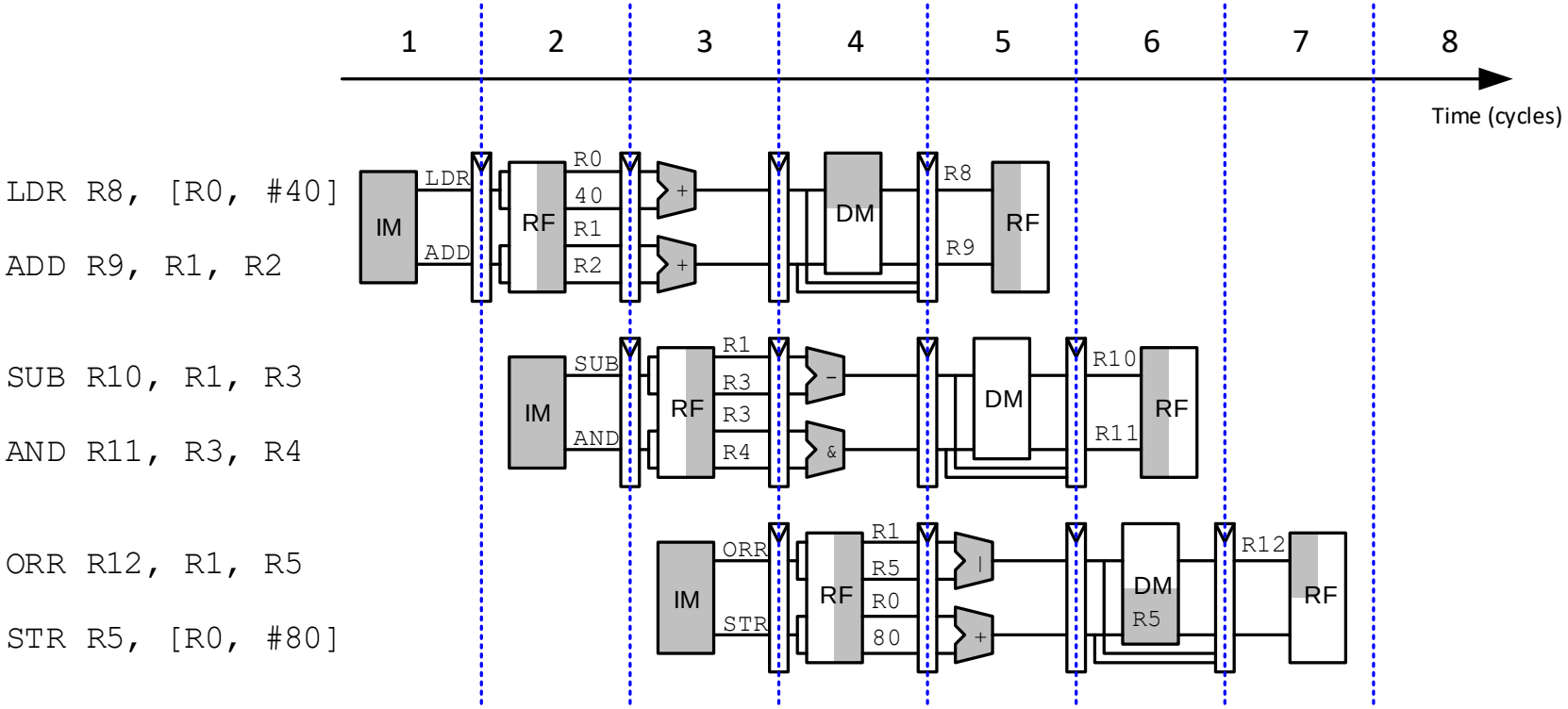
- Dynamic multiple-issue processors (aka superscalar)
  - Decisions on which instructions to execute simultaneously (usually in the range of 2 to 8) are being made dynamically (at run time by the hardware)
  - Most modern processors are superscalar
- Static multiple-issue processors (aka Very Large Instruction Word - VLIW)
  - Decisions on which instructions to execute simultaneously are being made statically (at compile time by the compiler)

Multiple-Issue Datapath Responsibilities

- Must handle, with a combination of hardware and software fixes, the fundamental limitations of
  - How many instructions to issue in one clock cycle – issue slots
  - Storage (data) dependencies – aka data hazards
    - Limitation more severe in a SS/VLIW processor due to (usually) low ILP
  - Procedural dependencies – aka control hazards
    - Ditto, but even more severe
    - Use dynamic branch prediction to help resolve the ILP issue
  - Resource conflicts – aka structural hazards
    - A SS/VLIW processor has a much larger number of potential resource conflicts
    - Functional units may have to arbitrate for result buses and register-file write ports
    - Resource conflicts can be eliminated by duplicating the resource or by pipelining the resource

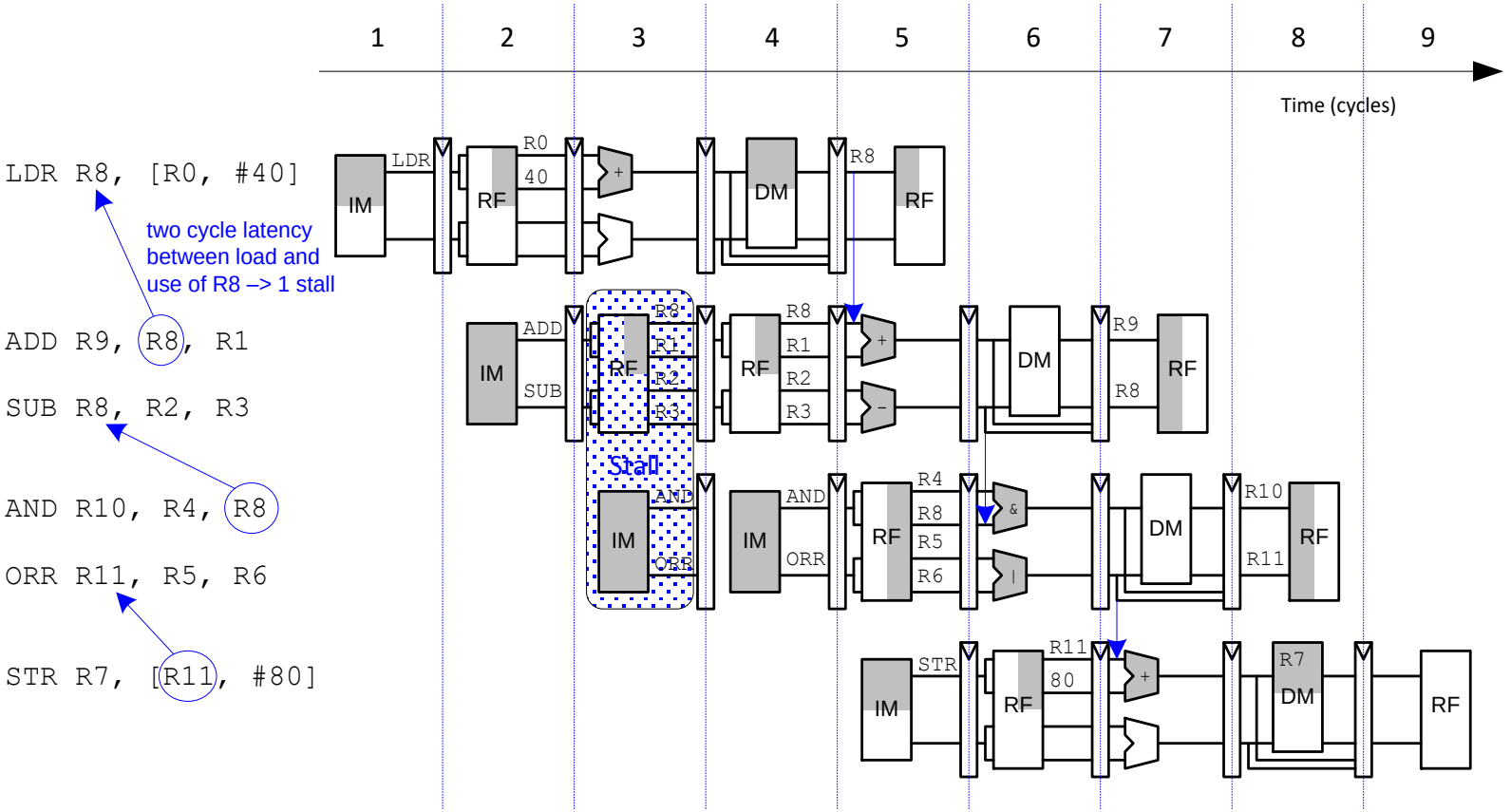
Superscalar – Ideal Case

- Ideal IPC = 2
- Actual IPC = 2



Superscalar – Practical Case

- Ideal IPC = 2
- Actual IPC = 1.2. Dependencies limit the achievable IPC



Out of Order Processor

- Looks ahead across multiple instructions
- Issues as many instructions as possible at once
- Executes instructions out of order (as long as no dependencies)
- Dependencies:
  - RAW (read after write, a.k.a true data dependency)
  - WAR (write after read, a.k.a antidependency)
  - WAW (write after write, a.k.a output dependency)

} storage conflicts

```
L1 : R3 = R3 * R5
L2 : R4 = R3 + 1
L3 : R3 = R5 + 1
```

Antidependency

True data dependency

Output dependency

If L3 executes before L1 / L2, L1 / L2 will use the value of R3 written by L3

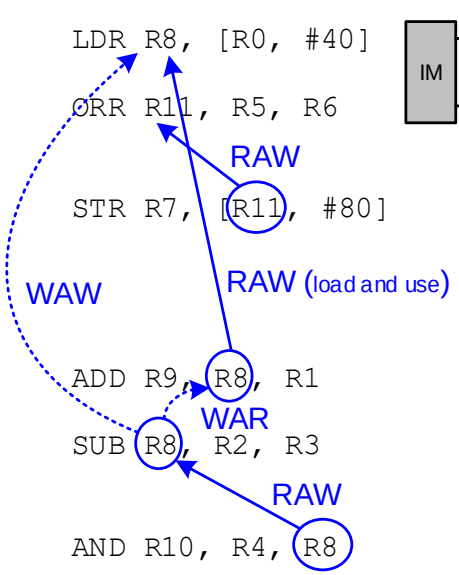
If L3 executes before L1, statements after L3 will use the value of R3 written by L1

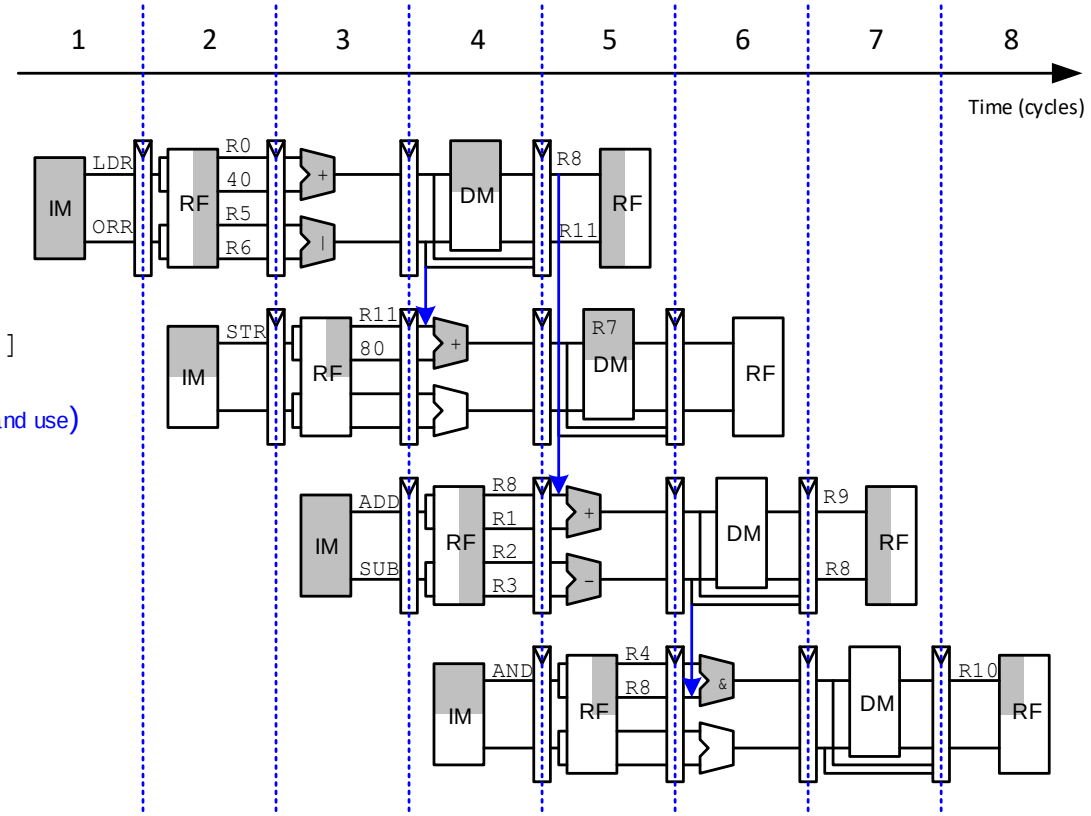
Out of order Example

```
LDR   R8, [R0, #40]
ADD   R9,  R8, R1
SUB   R8,  R2, R3
AND   R10, R4, R8
ORR   R11, R5, R6
STR   R7, [R11, #80]
```

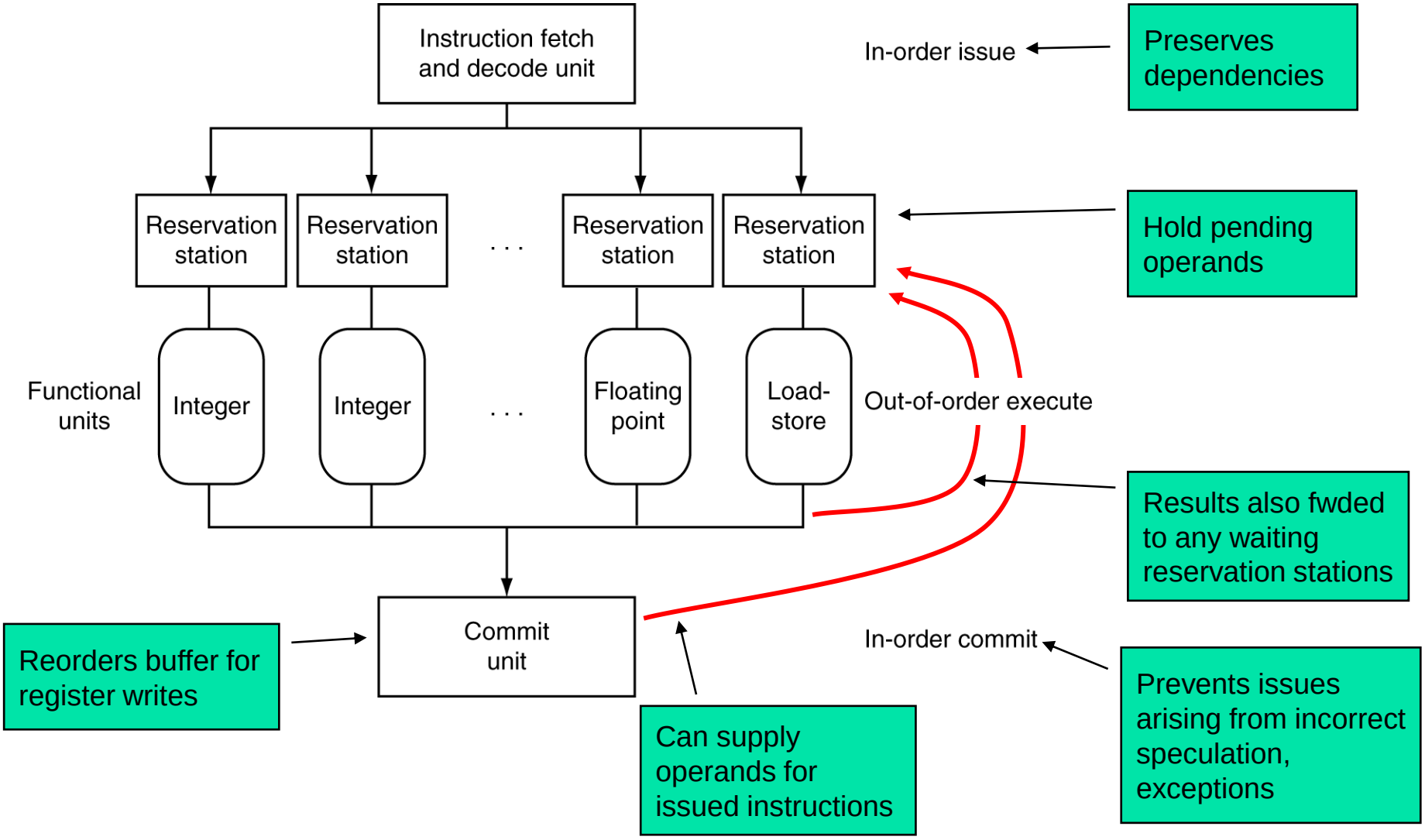
Ideal IPC : 2

Actual IPC : 6/4 = 1.5





Out of order Execution



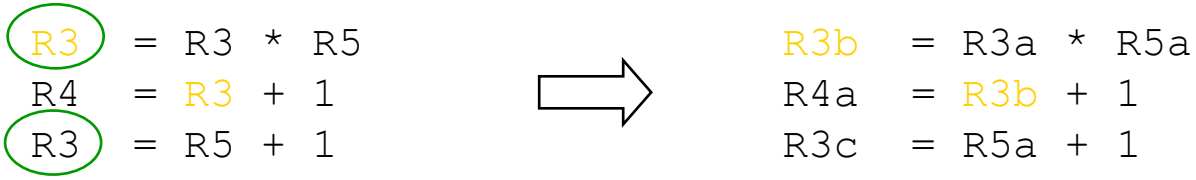
In-Order vs Out-of-Order

- Instruction fetch and decode units are required to issue instructions in-order so that dependencies can be tracked
- The commit unit is required to write results to registers and memory in program fetch order so that
  - if exceptions occur the only registers updated will be those written by instructions before the one causing the exception
  - if branches are mispredicted, those instructions executed after the mispredicted branch don't change the machine state (i.e., we use the commit unit to correct incorrect speculation)
- Although the front end (fetch, decode, and issue) and back end (commit) of the pipeline run in-order, the FUs are free to initiate execution whenever the data they need is available – out-of-(program) order execution
  - Allowing out-of-order execution increases the amount of ILP

Storage Conflicts and Register Renaming

- Storage conflicts can be reduced (or eliminated) by increasing or duplicating the troublesome resource
  - Provide additional registers that are used to reestablish the correspondence between registers and values
    - Allocated dynamically by the hardware in SS processors

- Register renaming – the processor renames the original register identifier in the instruction to a new register (one not in the ISA / visible register set)



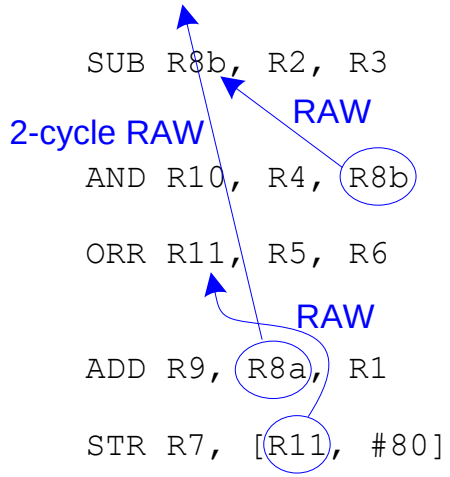
  - The hardware that does renaming assigns a "replacement" register from a pool of free registers and releases it back to the pool when its value is superseded and there are no outstanding references to it

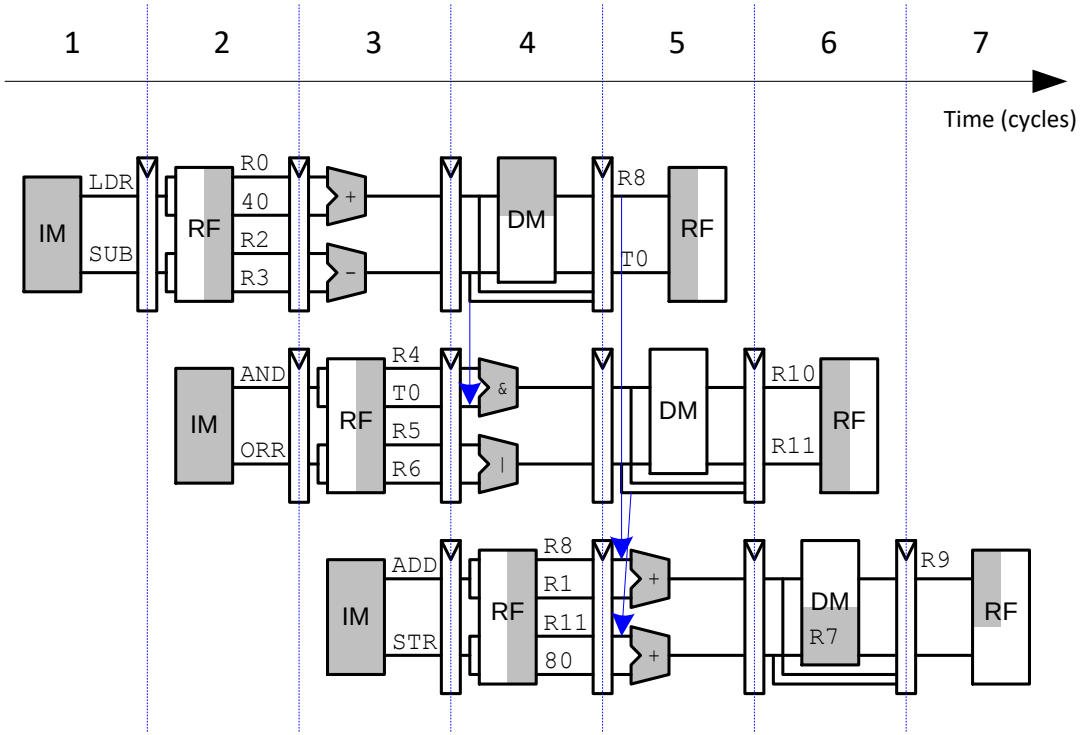
Register Renaming

```
LDR   R8, [R0, #40]
ADD   R9,  R8, R1
SUB   R8,  R2, R3
AND   R10, R4, R8
ORR   R11, R5, R6
STR   R7, [R11, #80]
```

Ideal IPC : 2

Actual IPC : 6/3 = 2





Speculative Execution

- Speculation is used to allow execution of future instructions that (may) depend on the speculated instruction
  - Speculate on the outcome of a conditional branch (branch prediction)
  - Speculate that a store (for which we don't yet know the address) that precedes a load does not refer to the same address, allowing the load to be scheduled before the store (load speculation)
- Must have (hardware and/or software) mechanisms for
  - Checking to see if the guess was correct
  - Recovering from the effects of the instructions that were executed speculatively if the guess was incorrect
- Ignore and/or buffer exceptions created by speculatively executed instructions until it is clear that they should really occur

VLIW Processors

- The compiler packs groups of independent instructions into the bundle
  - The bundle can be thought of as one very long instruction. Hence the name
- The compiler uses loop unrolling to expose more ILP
- Compile time register renaming (using visible register set) used to solve storage dependencies
- While superscalars use dynamic prediction, VLIW's primarily depend on the compiler for branch prediction
  - Compiler encodes prediction as a hint bit in the branch instruction format
  - Loop unrolling reduces the number of conditional branches
  - Predication (conditional execution) eliminates if-then-else branch structures by replacing them with predicated instructions

VLIW Processor Example …*

- Example : Intel Itanium and Itanium 2 for the IA-64 ISA – EPIC[$] (Explicit Parallel Instruction Computer)
  - 128-bit "bundles" containing three instructions, each 41-bits plus a 5-bit template field (specifying which FU each instruction needs)
  - Five functional units (IntALU, Mmedia, Dmem, FPALU, Branch), 128 integer registers, 128 floating point registers (82 bits long to preserve precision for intermediate results), …
  - Extensive support for speculation and predication
  - Intel hoped it would take off in server market, and gradually percolate into PC/consumer market as 64-bit processing becomes popular
  - In contrast, AMD just extended the original 32-bit instruction set to 64-bit (x86_64 a.k.a AMD64) -> No need to recompile 32-bit code
  - "The Itanium approach...was supposed to be so terrific—until it turned out that the wished-for compilers were basically impossible to write" - Donald Knuth

[$] Not to be confused with AMD's new Epyc x86_64 based server platform

VLIW vs Superscalar

- VLIW Advantages
  - Simpler hardware (potentially less power hungry)
  - Potentially more scalable - Allow more instructions per VLIW bundle and add more FUs
- VLIW Disadvantages
  - Programmer/compiler complexity and longer compilation times
  - Not all stalls are predicable - e.g., cache misses
  - Can't always schedule around branches - branch outcome is dynamically determined
  - Object (binary) code incompatibility – microarchitecture dependence
  - Needs lots of program memory bandwidth
  - Code bloat - `NOP`s need to be inserted if the compiler can't find independent instructions; Loop unrolling to expose more ILP uses more program memory space

Does Multiple Issue Work?

- Yes, but not as much as we'd like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well

Processes and Threads

- Process: program running on a computer
  - Multiple processes can run at once: e.g., surfing web, playing music, writing a report
- Thread: part of a program
  - Each process can have multiple threads: e.g., a word processor may have threads for typing, spell checking, printing etc., sharing the memory space (page table)
  - The operating system schedules threads
  - In a conventional processor, one thread runs at a time
  - In a multi-core / processor system, the number of threads running can be equal to the number of cores
  - In both the above cases, it appears to user that all threads running simultaneously

Threads in a Conventional Processor

- In *co-operative multitasking*, when one thread stalls (for example, waiting for I/O or similar events):
  - Architectural state of that thread stored
  - Architectural state of waiting thread loaded into processor and it runs
  - Called context switching
  - The next (ready) thread to be run determined by the OS scheduler
- In *preemptive multitasking*, the context switch can also be initiated through timer interrupts, without having to wait for the thread to stall

Hardware Multithreading

- Processor maintains multiple copies of architectural state (PC, registers, xPSR etc.)
  - OS sees it as multiple cores and schedules multiple threads
  - When one thread stalls for minor events such as cache misses, another runs immediately, without the OS initiating a context switch
  - This is called *coarse-grained multithreading*
- Does not increase instruction-level parallelism (ILP) of single thread, but increases throughput
  - Exploits thread-level parallelism

Hardware Multithreading

- Fine grained multithreading (barrel processor)
  - Executes one instruction from each thread in alternate clock cycles
  - Spaces instructions from a single thread by at least one instruction, thereby reducing dependencies and branch misprediction penalties as well as the hardware to deal with those
- Simultaneous multithreading (SMT a.k.a Hyperthreading)
  - If one thread can't keep all execution units busy, another thread can use them
  - Instructions from different threads execute at the same time, without duplication of functional units
  - Intel claims ~25% improvement in performance using less than 5% extra hardware
  - OS needs to be SMT-aware. Why?
- Coarse and fine-grained MT are temporal, SMT has both spatial and temporal aspects