

# **Instruction Level Parallelism**

Tricking the system to maximize throughput

# Source material

- D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition*. Cambridge, MA: Morgan Kaufmann, 2018.
  - Chapter 4.7: Data Hazards: Forwarding versus Stalling
  - Chapter 4.8: Control Hazards
  - Chapter 4.10: Parallelism via Instructions
- J. L. Hennessy, D. A. Patterson, and K. Asanović, *Computer Architecture: A Quantitative Approach*, 5th ed. Waltham, MA: Morgan Kaufmann/Elsevier, 2012.
  - Appendix C: Pipelining: Basic and Intermediate Concepts
  - Chapter 3: Instruction-Level Parallelism and Its Exploitation

# Goal: Improving throughput by increasing ILP

1. Revisit how we circumvent hazards to avoid pipeline stalls
2. More hardware techniques to improve throughput
3. Software techniques to improve throughput

# Revisiting hazards

When architects became less conservative

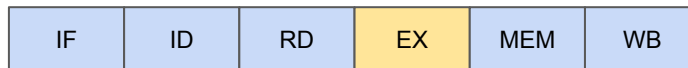
# Revisiting data hazards: Types of dependencies

- Read After Write (RAW)

add R1, R1, #1



add R3, R1, R2



- Write After Read (WAR)

add R3, R1, R2



load R1, (R2)



- Write After Write (WAW)

add R1, R1, #1



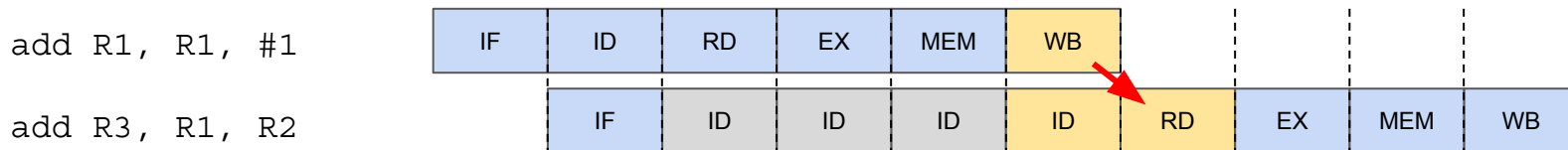
load R1, (R2)



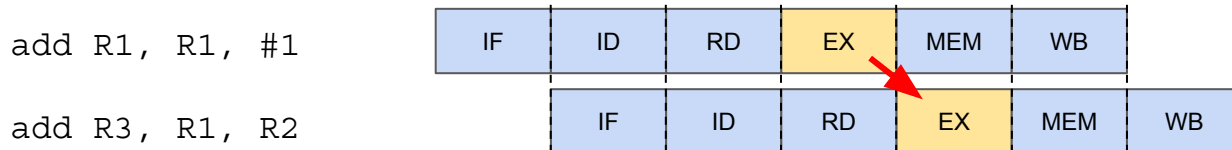
# Revisiting data hazards: Bypassing

- Result of first instruction is produced at the output of stage EX
- Source of second instruction is required at the input of stage EX
- Limiting factor: source operands can only be read from register file (after stage WB)
- Workaround: add a path that forwards/bypasses data from producer to consumer

- Without bypass → Stall three cycles



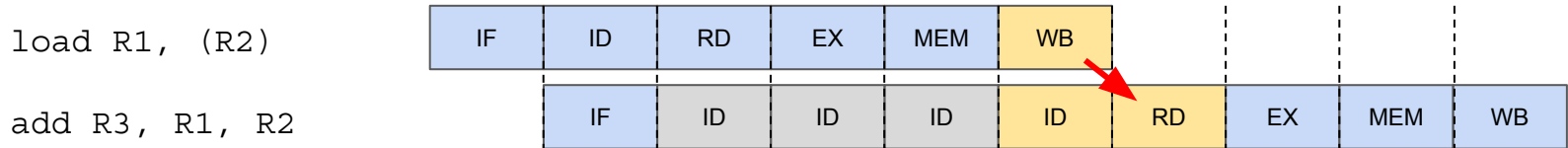
- With bypass → No stalls



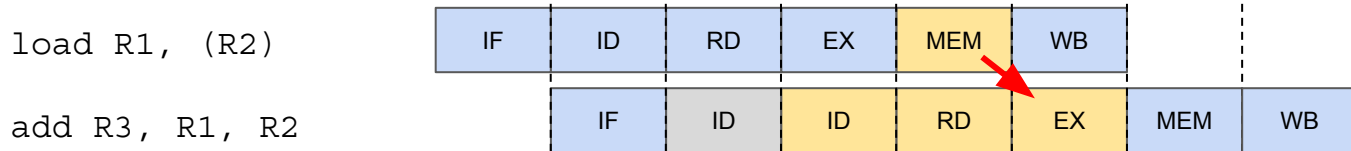
# Revisiting data hazards: Bypassing (cont.)

- Bypassing helps reducing pipeline stalls due to data hazards
- Not all stalls can be avoided

- Without bypass → Stall three cycles



- With bypass → Stall one cycle



# Revisiting control hazards: Branch prediction

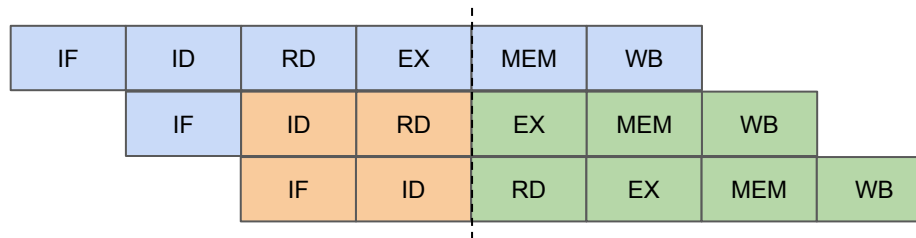
- Instead of stalling the pipeline, we can assume that branches are never taken

- Prediction correct → No stalls

beq R1, myfunc

add R3, R1, R2

mul R3, R3, #2

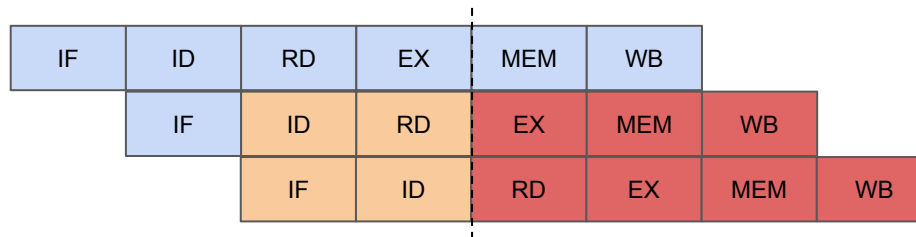


- Prediction incorrect → Need to kill the instructions in-flight!!

beq R1, myfunc

add R3, R1, R2

mul R3, R3, #2



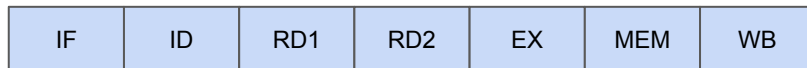


# Revisiting control hazards: Branch prediction (cont.)

- There are two aspects which need to be predicted
  - If branch is taken or not
  - What is the target instruction of the branch
- Branch prediction is a very active topic in computer architecture research
- For this course you need to know
  - Branch prediction exists and has an impact on instruction throughput
  - Execution based on branch prediction is called Speculative Execution
  - Speculative execution requires a mechanism to kill in-flight instructions
- If you want to learn more
  - Chapter 3.3 of Computer Architecture: A Quantitative Approach
  - Look into the proceedings of almost any computer architecture conference

# Revisiting structural hazards: Adding resources

- Example: Reading two operands from the register file during the same cycle
  - Limitation: My register file only has one read port
  - Current implementation: Spend two cycles reading operands



- Workaround: Add another read port
- Benefit: Only one cycle to read operands
- Drawback: More expensive design

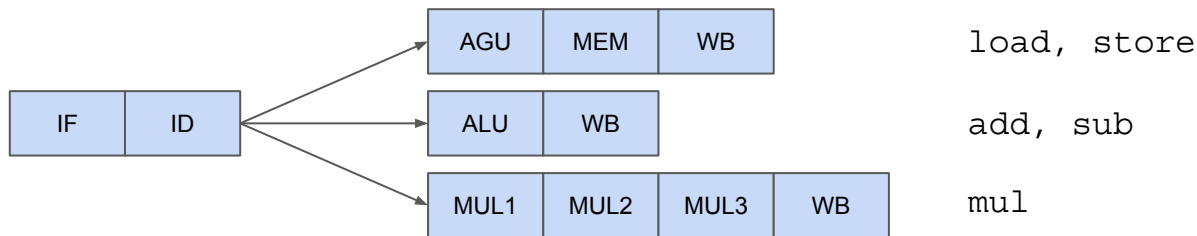


# **Multi-cycle pipelines**

When architects decided to diversify

# Different stages for different instructions

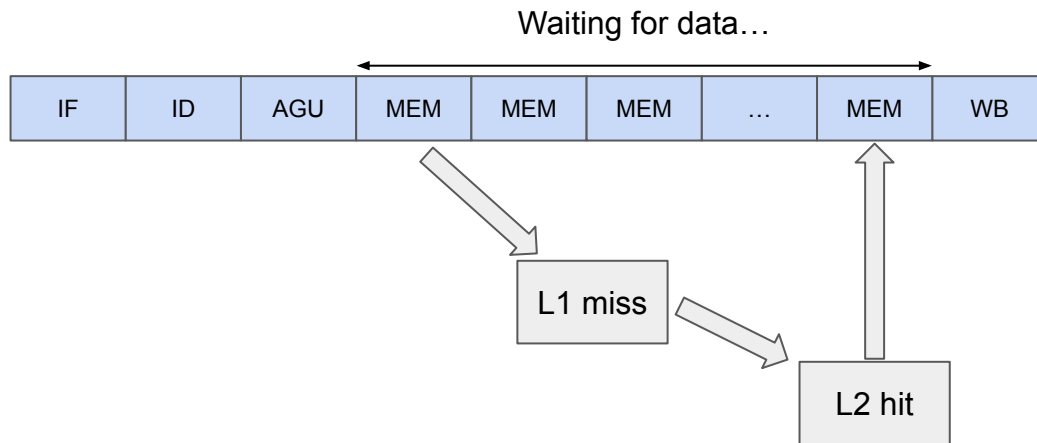
- Until now, all instructions follow the same pipeline
- Arithmetic instructions go through the MEM stage without it being necessary
- Proposal: Split pipeline into multiple paths and choose one path after decoding
- Also allows to separate costly operations (eg. multiplications) into their own pipeline



- Benefit: Reducing latency of some operations
- Disclaimer: We are still issuing one instruction per cycle and preserving program order

# Number of cycles is not always known at decode

- In micro-architectures that include caches, how many cycles will a load cost?
  - L1 Hit → 1 cycle? 3 cycles?
  - L1 Miss, L2 Hit → 20 cycles?
  - Going to main memory → 100 cycles?
- Special logic has to be added to the MEM stage to wait for load data to arrive



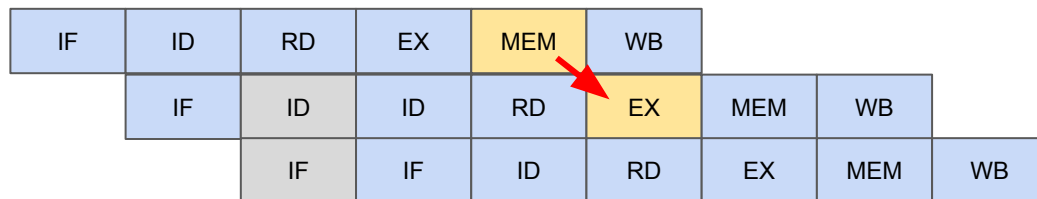
# Dynamic Scheduling

When architects decided to play God

# Revisiting data hazards: Beyond bypassing

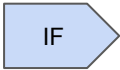
- Not all data hazards can be circumvented with bypassing
- Pipeline stalls may block later instructions that do not have a dependency

```
1. load R1, (R2)
2. add  R3, R1, R2
3. add  R2, #1
```



- There is a data dependency between 1 and 2
- There is NO dependency between 3 and any other previous instruction
- Why should 3 be stalled??
- Proposal: Issue 3 while 2 is waiting for the dependency to be resolved

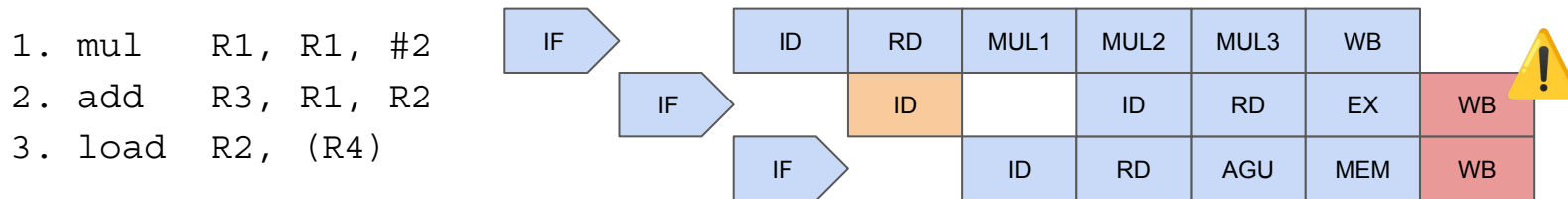
# Out-of-Order execution

- Deciding to alter the order of instruction execution is called Dynamic Scheduling
- Another common word for this technique is Out-of-Order execution (OoO)
- OoO allows us to circumvent data dependencies that cause data hazards
- OoO does NOT circumvent other hazards (eg. structural hazards)
- Fetched instructions are placed into a queue or buffer waiting to be picked by the dynamic scheduler 
- When an instruction finishes execution (result available), it has completed
- When the instruction result is written and cannot be undone, it has been retired

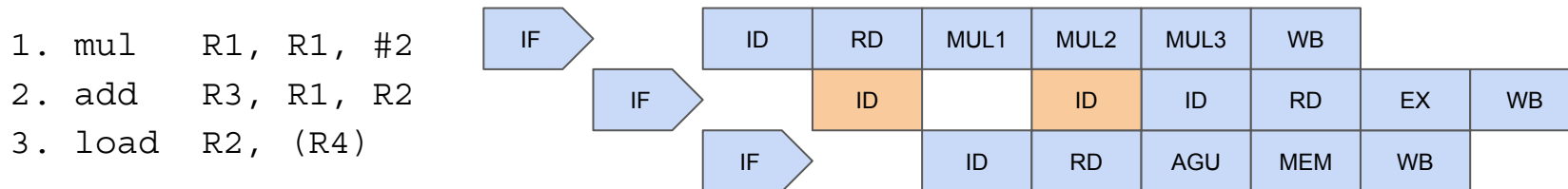


# Out-of-Order execution: New headaches

- Upon decoding 2, a data hazard is discovered and the instruction is stalled
- Next cycle, 3 is decoded and issued since it does not have a data dependency



- Sadly, 2 and 3 try to WB during the same cycle → Structural hazard
- Correct OoO execution must stall 2 an extra cycle to avoid the hazard



# Out-of-Order execution: Revisiting dependencies

- In an in-order pipeline, the only data dependency that could cause a hazard was a true dependence (RAW)
- OoO execution introduces the possibility of
  - Anti-dependence (WAR)
  - Output dependence (WAW)

# New issues → New solutions: Register Renaming

- Proposal: Implement more physical registers than the ISA defines
  - Logical registers (ISA-defined): R0 - R31
  - Physical registers (implementation-defined): P0 - P63
- The decoding logic renames the logical registers of all instructions to physical registers
- Each write to a logical register produces a new physical register translation

mul	<span style="border: 1px solid red; padding: 2px;">R1</span>	<span style="border: 1px solid purple; padding: 2px;">R1</span>	#2		mul	P1,	<span style="border: 1px solid purple; padding: 2px;">P0</span>	#2
...				→	...			
add	<span style="border: 1px solid green; padding: 2px;">R1</span>	<span style="border: 1px solid red; padding: 2px;">R1</span>	R3		add	<span style="border: 1px solid green; padding: 2px;">P2</span>	<span style="border: 1px solid red; padding: 2px;">P1</span>	P3

- Register renaming eliminates the WAR and WAW hazards
- If you want to know more, read about *Tomasulo's Algorithm*

# Out-of-Order + Speculative execution =

- In-order speculative execution introduced the need for a killing mechanism
- With OoO, speculative instructions can complete before the prediction is even resolved
- How can we undo writes to registers and/or memory?
- What about exceptions? (completely outside the scope of this course)
- There is now easy answer...
  - For registers, register renaming helps
  - For memory, some approaches implement Load and Store Buffers (LB, SB) that hold petitions from/to memory until the prediction is resolved

# Superscalar Pipelines

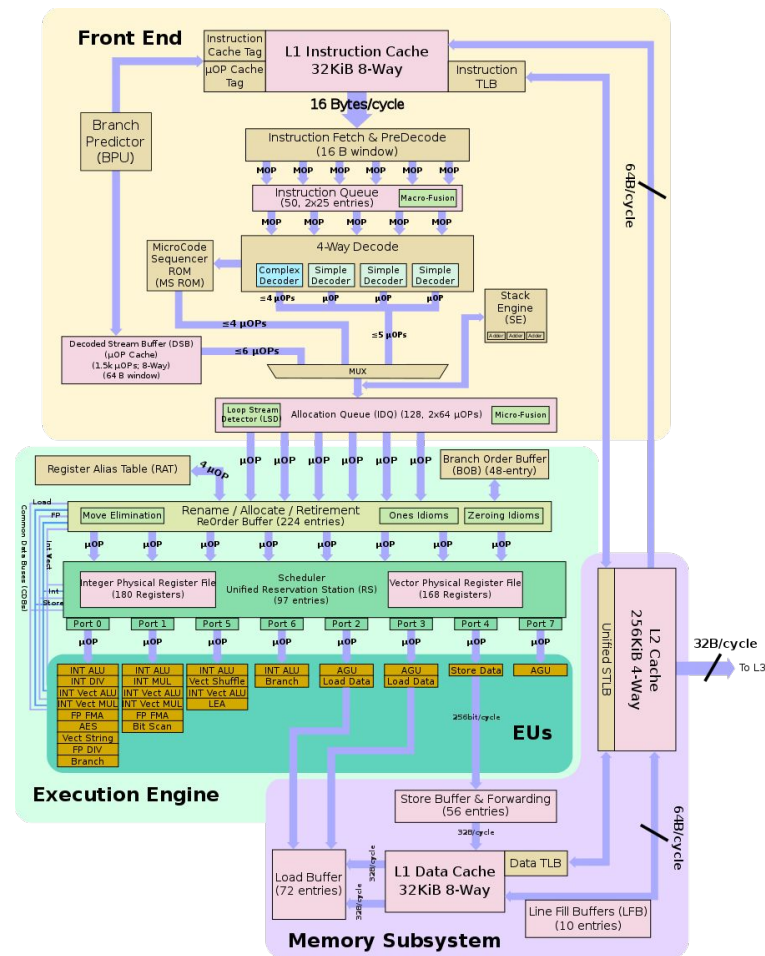
Beyond  $IPC = 1$

# Revisiting multi-cycle pipelines

- Having implemented:
  - Multi-cycle pipeline (different pipelines for different types of instructions)
  - Out-of-Order execution
- Why stop at one issue per cycle?
- If no hazards → issue logic can feed into one or more pipelines on a single cycle
- When issuing more than one instruction, we say that it is a Superscalar Pipeline
- The benefit is limited if the pipeline can only retire one instruction per cycle
- Modern processors are able to issue and retire more than one instruction per cycle

## Real use case: Intel Skylake

- Front-End
  - Out-of-Order execution
  - Multiple fetch per cycle
  - Register renaming
- Multiple issue per cycle
- Back-End
  - Eight execution pipelines
  - Load Buffer with 72 entries
  - Store Buffer with 56 entries



# Calculating theoretical FP peak

## ■ Arithmetic (eg. Floating-Point) performance

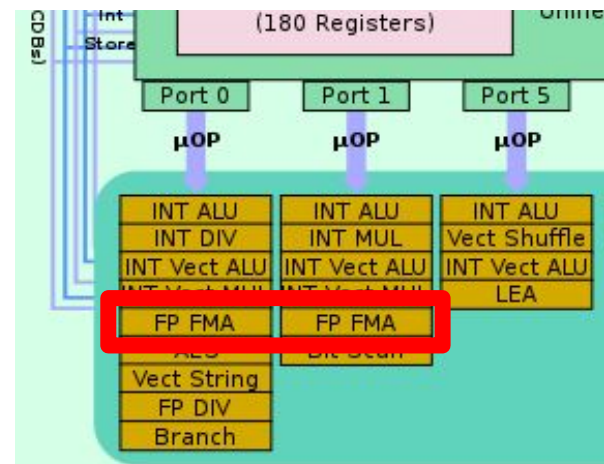
- P := Number of Pipelines
- U := Number of Functional units (for SIMD, width of SIMD registers)
- T := Throughput of instruction in ONE pipeline
- O := Operations per instruction
- F := CPU frequency

## ■ Real use case: Intel Skylake

- P = 2 FMA pipelines
- U = 8 DP elements per SIMD register (AVX512)
- T = 1 FMA per cycle
- O = 2 operations per instruction
- F = 2.10 GHz (cycles / second)

## ■ Peak performance

- ~~2 \* 8 elements/instruction \* 1 instruction/cycle \* 2 ops/element~~ = 32 ops/cycle
- 32 ops/cycle \* 2.10 \* 10<sup>9</sup> cycles/second = 67.20 \* 10<sup>9</sup> ops/s = **67.20 GFlop/s**



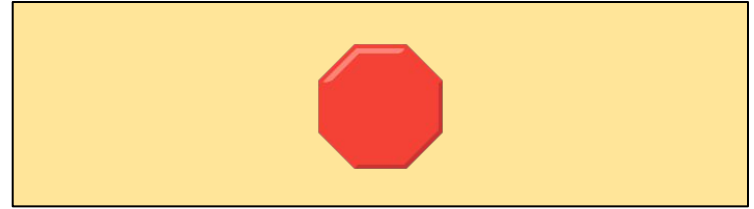
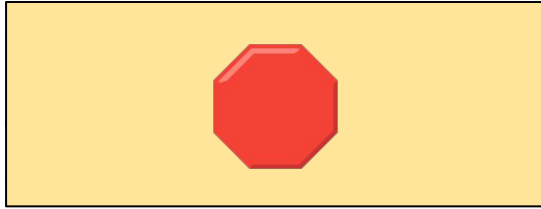


# Another example: Cortex-A57 (Jetson TX1)

- <https://chipsandcheese.com/p/cortex-a57-nintendo-switchs-cpu>

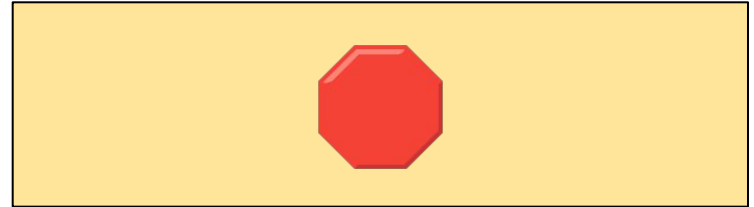
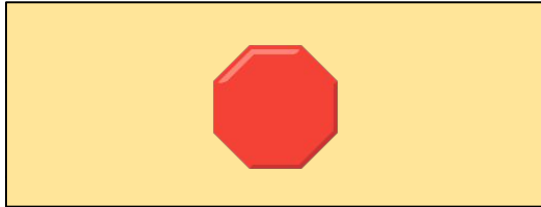
- DP Scalar instructions (aarch64)

- P =
- U =
- T =
- O =



- DP SIMD instructions (NEON)

- P =
- U =
- T =
- O =



# **Software techniques to improve ILP**

When programmers try to help architects

# Basic block

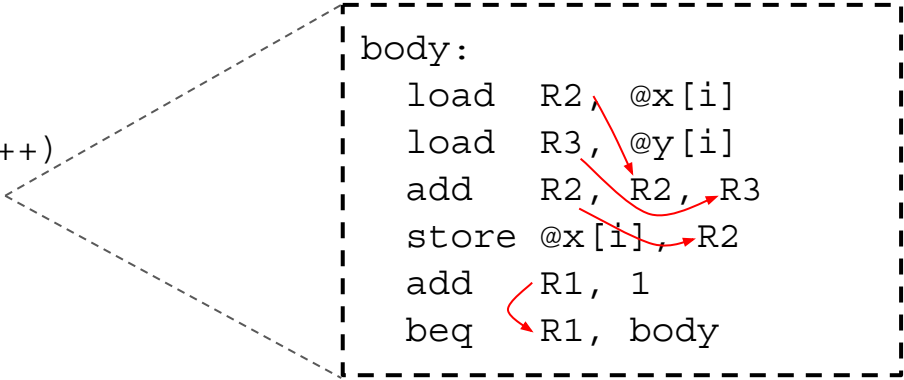
*Straight-line code sequence with no branches in except to the entry and no branches out except at the exit.*

Chapter 3 of Computer Architecture: A Quantitative Approach

- Sequence of instructions that are likely to depend upon each other
- Small blocks with lots of dependencies limit the overlap between instructions

- Example: Body of a loop

```
for (i=0; i < 1000; i++)  
    x[i] = x[i] + y[i]
```



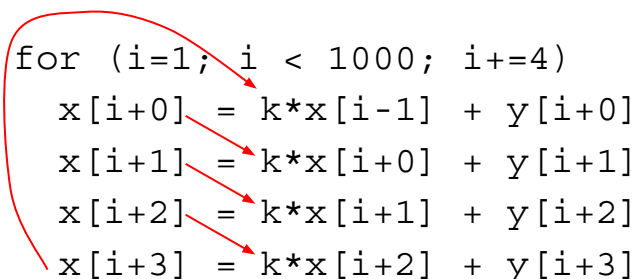
```
body:  
    load    R2, @x[i]  
    load    R3, @y[i]  
    add     R2, R2, R3  
    store   @x[i], R2  
    add     R1, 1  
    beq     R1, body
```

1. Increase size of basic block
2. Leverage Instruction Level Parallelism (ILP) across multiple basic blocks

# Loop unrolling

- Technique to increase the basic block size
  - Benefit: Exposes more ILP
  - Drawbacks: Less instruction cache locality, increased register pressure
- Limitations
  - If there are dependencies across iterations, there is little to be gained
  - Number of iterations must be multiple of the unroll factor

```
for (i=0; i < 1000; i+=4)
    x[i+0] = k*x[i+0] + y[i+0]
    x[i+1] = l*x[i+1] + y[i+1]
    x[i+2] = k*x[i+2] + y[i+2]
    x[i+3] = k*x[i+3] + y[i+3]
```

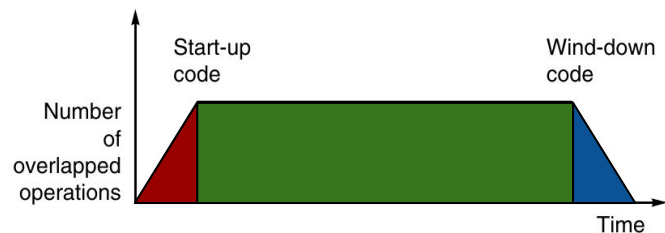


```
for (i=1; i < 1000; i+=4)
    x[i+0] = k*x[i-1] + y[i+0]
    x[i+1] = k*x[i+0] + y[i+1]
    x[i+2] = k*x[i+1] + y[i+2]
    x[i+3] = k*x[i+2] + y[i+3]
```

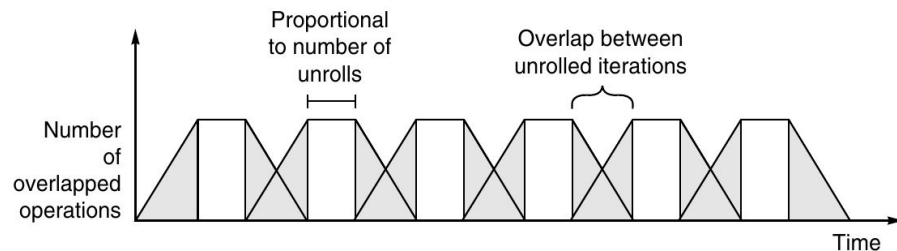
The diagram illustrates the unrolled loop with a red circle around the first iteration and red arrows pointing from the right-hand side of one iteration to the left-hand side of the next, highlighting the data dependencies between iterations.

# Software pipelining

- Reorganize loops such that each iteration contains instructions from different iterations of the original code.
- Very tricky to implement → Let the compiler handle it



(a) Software pipelining



(b) Loop unrolling

```
load  R1, @k
```

```
load  R2, @x[i+0]
```

```
load  R3, @x[i+0]
```

```
mul   R4, R1, R2
```

```
add   R5, R4, R3
```

```
store @x[i+0], R5
```

```
load  R6, @x[i+1]
```

```
load  R7, @x[i+1]
```

```
mul   R8, R1, R6
```

```
add   R9, R8, R7
```

```
store @x[i+0], R9
```

...

# Remarks about software techniques

## 1. Simplify code structure

- Minimize conditional and nested regions of code
- Be aware of (and avoid) unnecessary data dependencies

## 2. Don't overdo it

- Some optimizations (eg. software pipelining) can be done by the compiler

## 3. Hardware implementation is smarter than you think

- Dynamic scheduling is able to do runtime re-ordering, which is not possible when writing code (not even from the compiler's point of view)
- There are many other hardware optimization techniques that we have not covered today!