MONASH University
Information Technology

FIT3143

LECTURE WEEK 8

PART I

INTRODUCTION TO INSTRUCTION LEVEL PARALLELISM (ILP) PROCESSORS

Overview

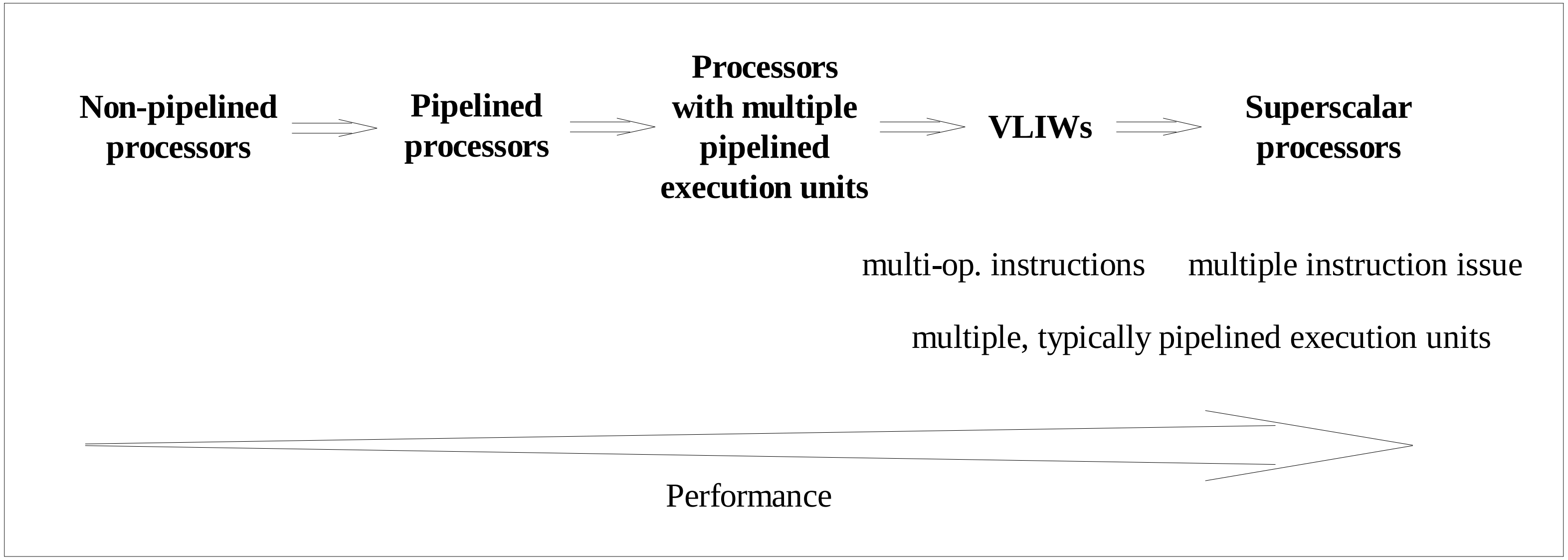
- Evolution and overview of ILP Processors
- Understanding of inter-instruction dependencies
  - Data, Control and Resource Dependencies
- What is instruction Scheduling?
- Sequential Consistency
- How fast can we go?

Learning outcome(s) related to this topic

- Explain the fundamental principles of parallel computing architectures and algorithms (LO1)

Evolution of ILP Processors

- Increasing processor performance has been provided by increased
Instruction Level Parallelism (ILP)



Non-pipelined processors → Pipelined processors → Processors with multiple pipelined execution units → VLIWs → Superscalar processors

multi-op. instructions

multiple instruction issue

multiple, typically pipelined execution units

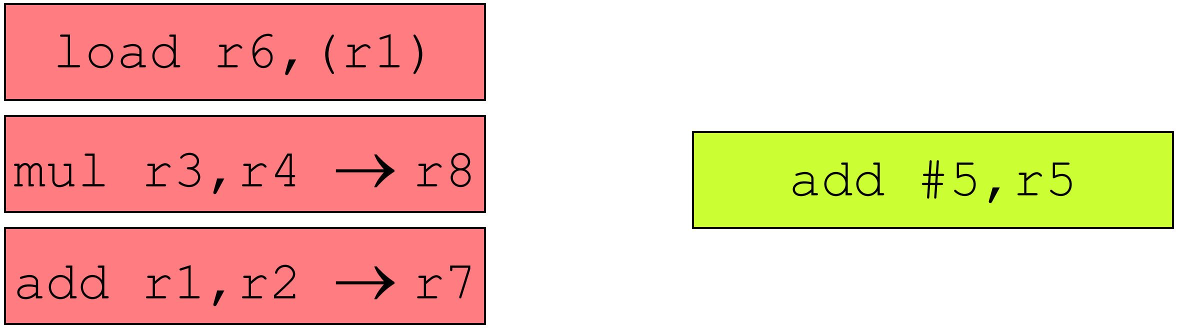
Performance

Instruction Level Parallelism

- Consider following code fragment

```
mul     r3, r4 → r8
add     r1, r2 → r7
load    r6,(r1)
add     #5, r5
```

- Which instructions are unrelated?
  - Can be executed in any order



load r6,(r1)

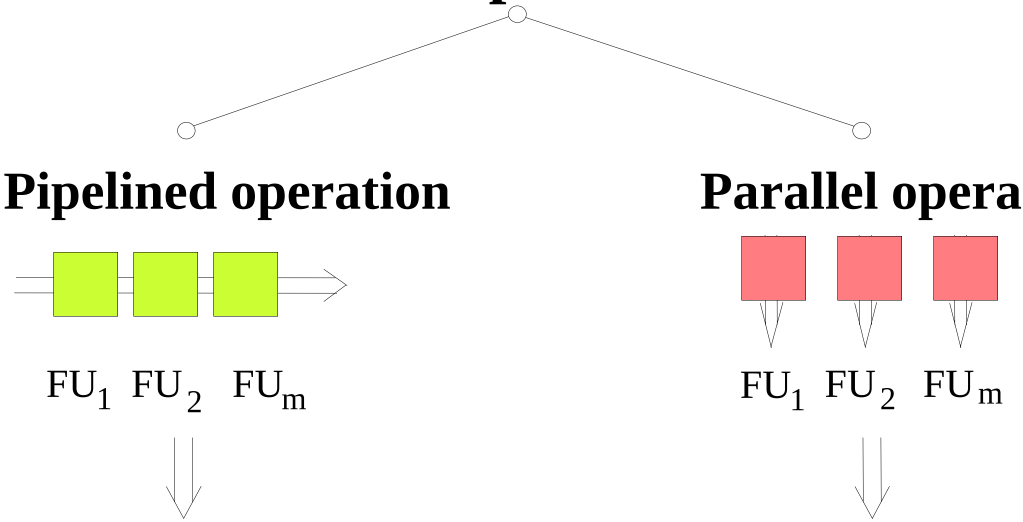
mul r3,r4 → r8

add r1,r2 → r7

add #5,r5

Pipelining or parallel?

Internal operating principle of IFP processors



Pipelined operation

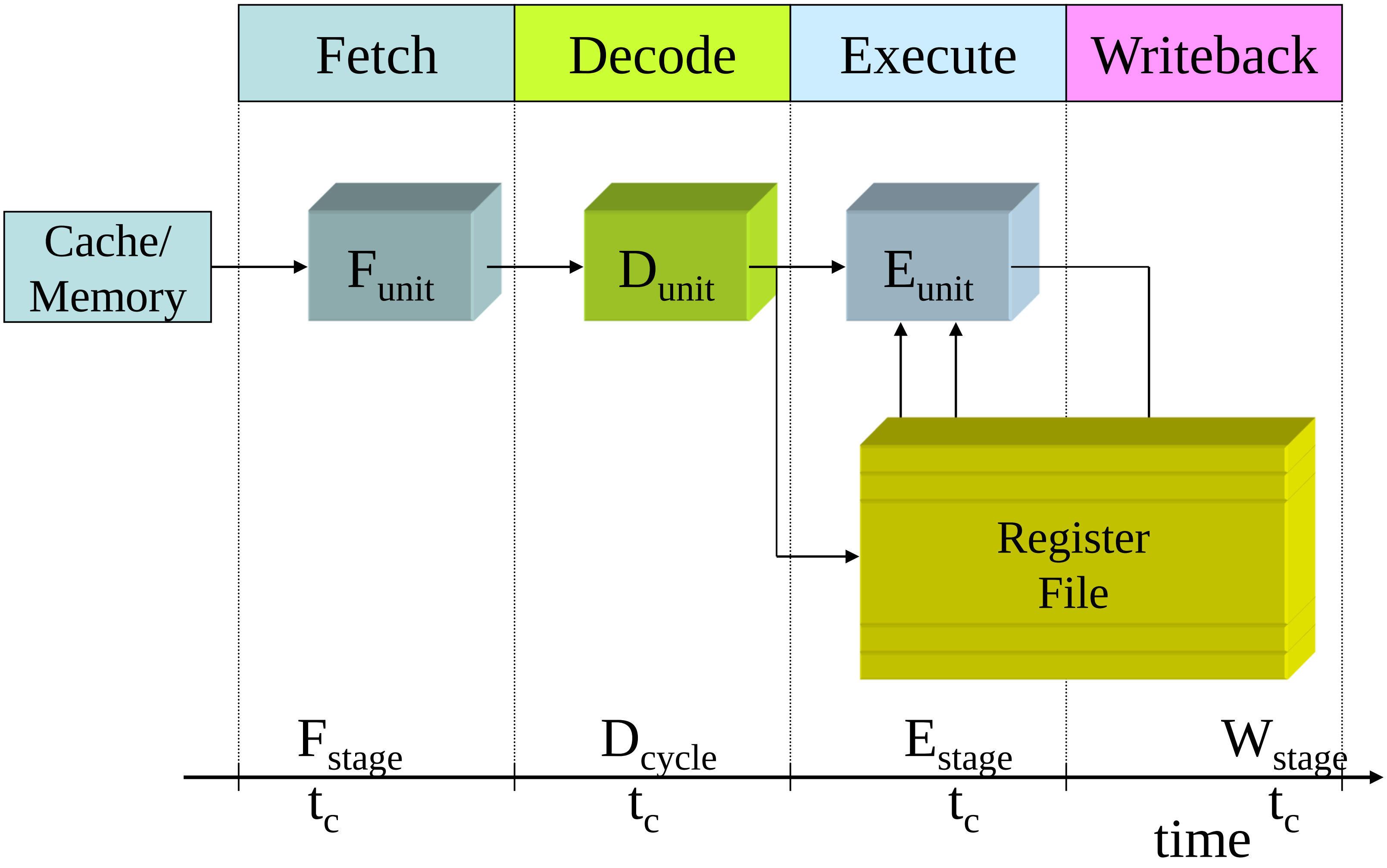
Parallel operation

Pipelined processors

**Very Long Instruction Word (VLIW) and
Super scalar processors**

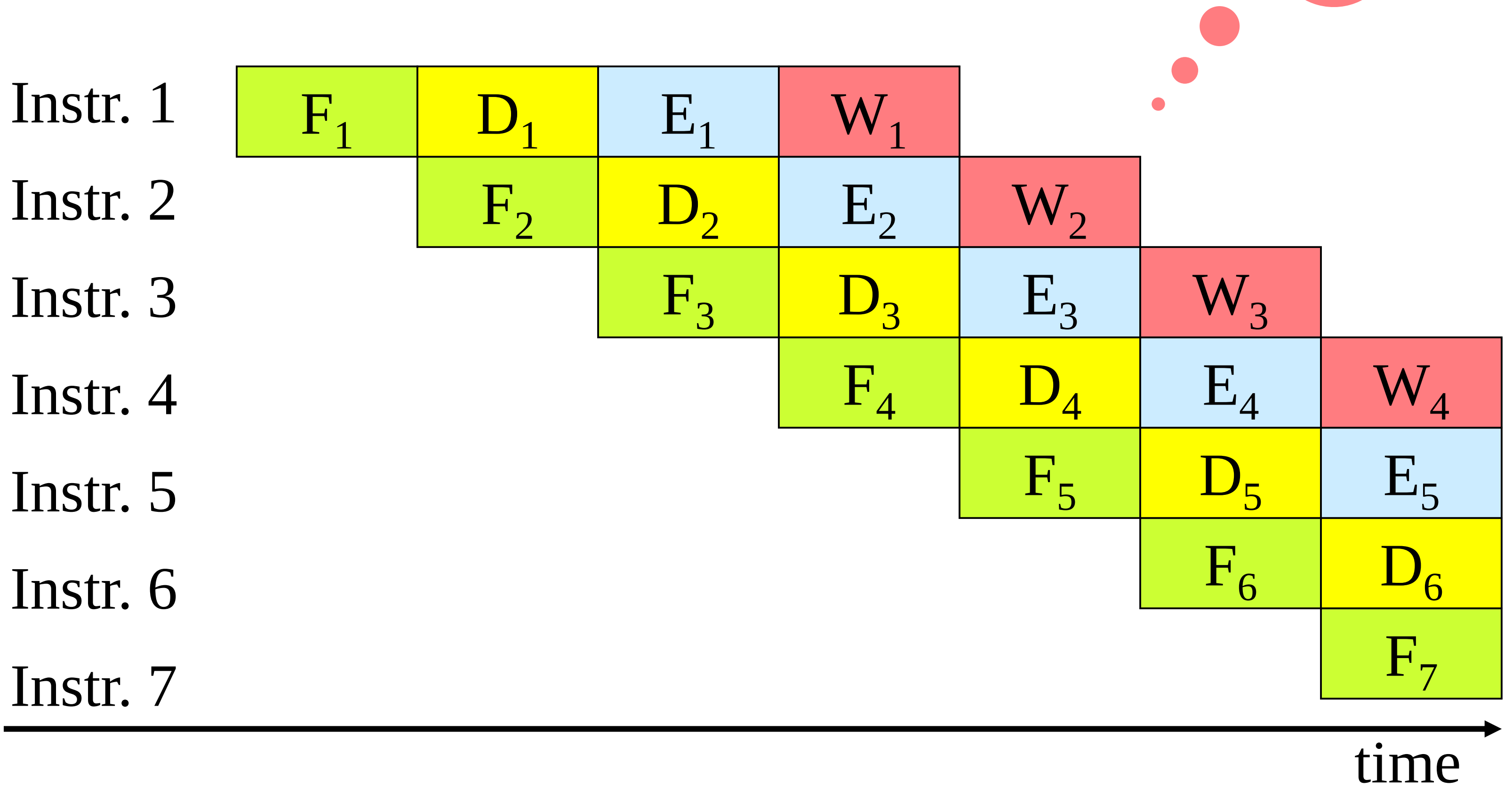
FU: functional unit

Pipelining


Fetch
Decode
Execute
Writeback
Cache/ Memory
F_{unit}
D_{unit}
E_{unit}
Register File
F_{stage}
D_{cycle}
E_{stage}
W_{stage}
t_c
t_c
t_c
t_c
time

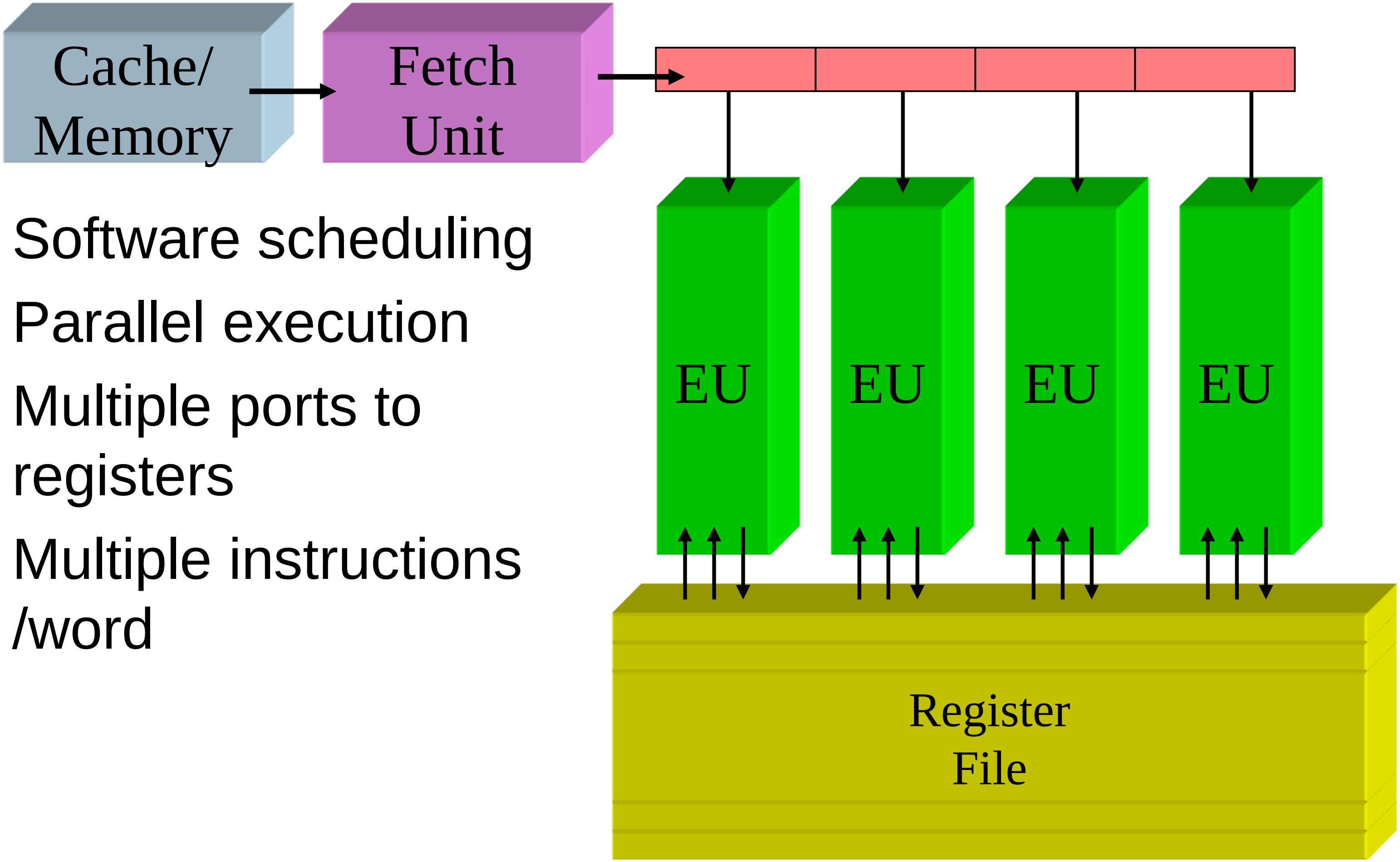
Machine Utilisation with pipelining

Machine
BUSY



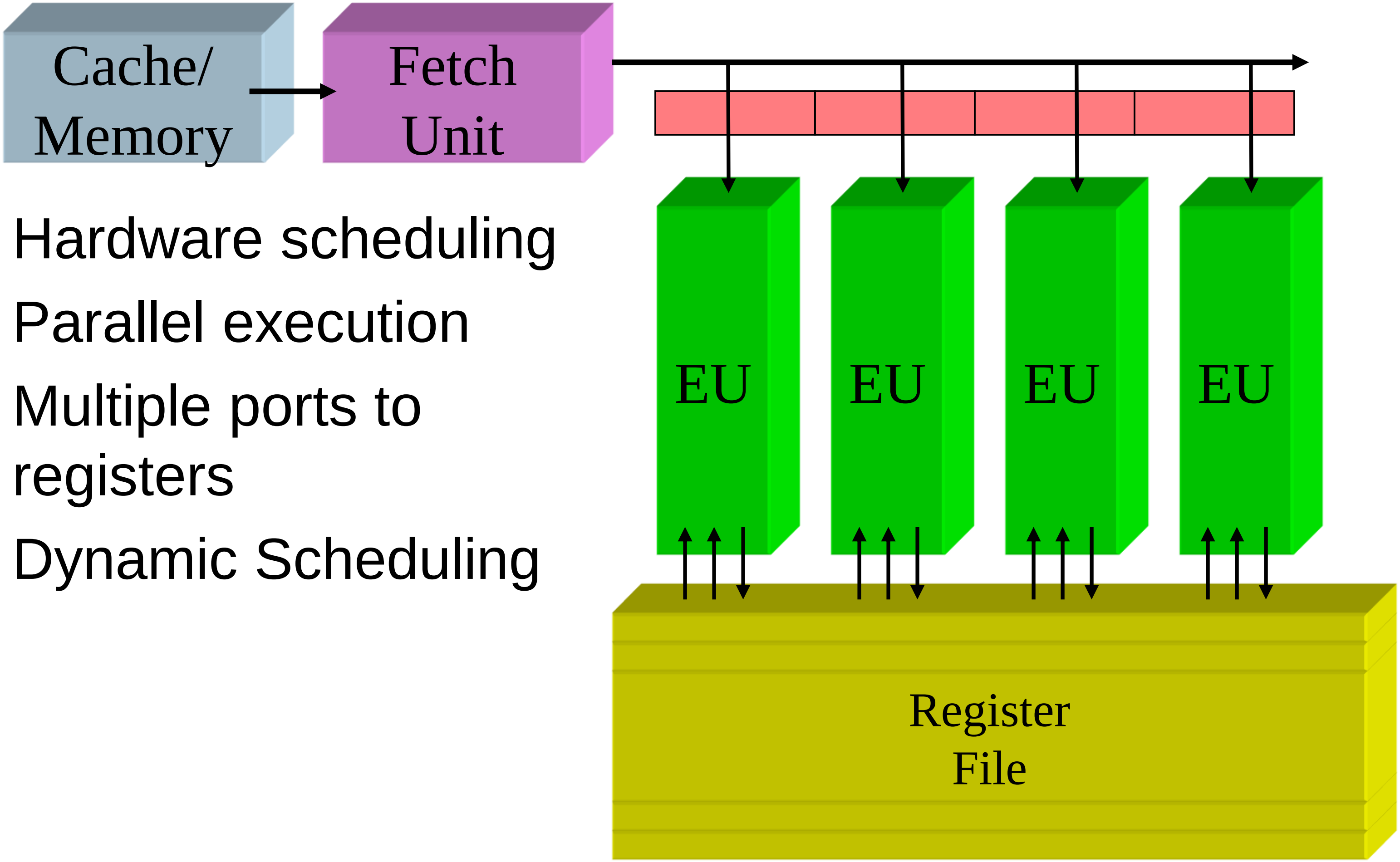
Instr. 1	F_1	D_1	E_1	W_1			
Instr. 2		F_2	D_2	E_2	W_2		
Instr. 3			F_3	D_3	E_3	W_3	
Instr. 4				F_4	D_4	E_4	W_4
Instr. 5					F_5	D_5	E_5
Instr. 6						F_6	D_6
Instr. 7							F_7

VLIW Approach



- Software scheduling
- Parallel execution
- Multiple ports to registers
- Multiple instructions /word

Superscalar Approach



- Hardware scheduling
- Parallel execution
- Multiple ports to registers
- Dynamic Scheduling

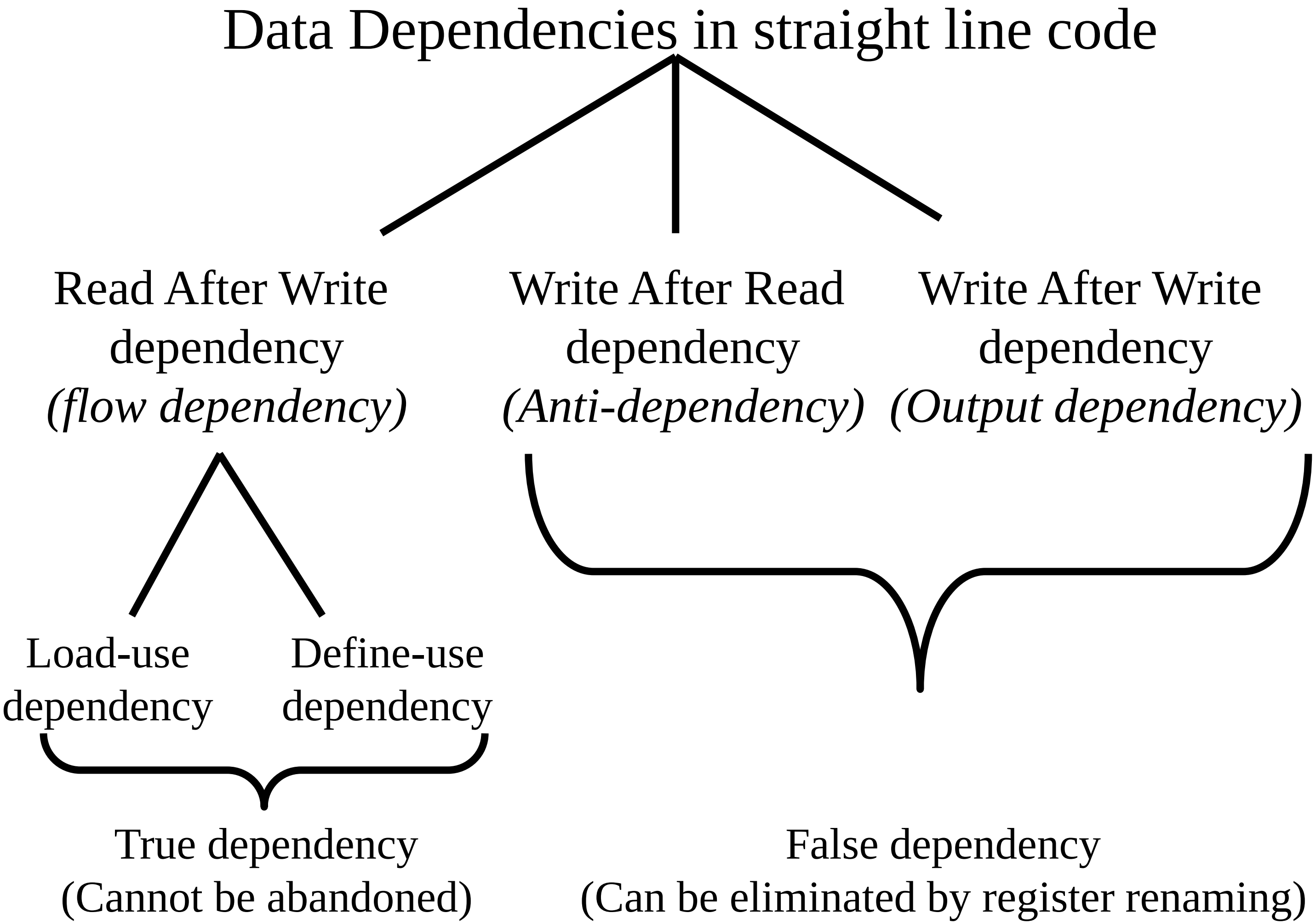
Out-of-order Execution (OOOE)

- Look ahead in a window of instructions and find instructions that are *ready to execute*
  - Don’t depend on data from previous instructions still not executed
  - Resources are available
- Out-of-order execution
  - Start instruction execution before execution of a previous instructions
- Advantages:
  - Help exploit Instruction Level Parallelism (ILP)
  - Help cover latencies (e.g., L1 data cache miss, divide)
- Can Compilers do the work ?
  - Compilers can *statically reschedule instructions*
  - Compilers do not have *run time information*
    - Conditional branch direction → limited to basic blocks
    - Data values, which may affect calculation time and control
    - Cache miss / hit

Dependencies between instructions

- Data Dependencies
  - Future instructions depend on results of prior ones
  - Registers and Memory
- Control Dependencies
  - Branch dependencies
- Resource Dependencies
  - Number of ALUs, memory ports, etc

Data Dependencies



Data Dependencies in straight line code

Read After Write dependency *(flow dependency)*

Write After Read dependency *(Anti-dependency)*

Write After Write dependency *(Output dependency)*

Load-use dependency

Define-use dependency

True dependency (Cannot be abandoned)

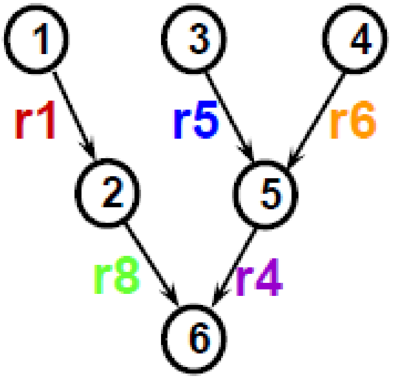
False dependency (Can be eliminated by register renaming)

Data Flow Analysis

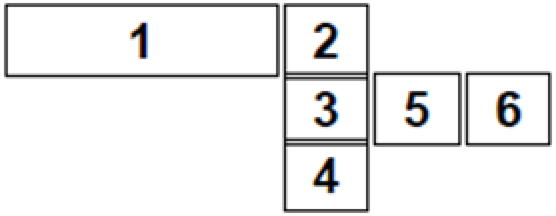
- Example

```
(1) r1 ← r4 / r7 ;  assume divide takes 20 cycles
(2) r8 ← r1 + r2   ; r1 depends on (1)
(3) r5 ← r5 + 1    ; r5 is independent
(4) r6 ← r6 - r3   ; r6 & r3 are independent
(5) r4 ← r5 + r6   ; r5 depends on (3); r6 depends on (4)
(6) r7 ← r8 * r4   ; r8 depends on (2); r4 depends on (5)
```

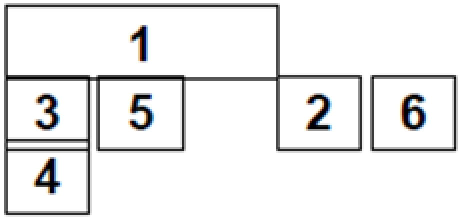
Data Flow Graph



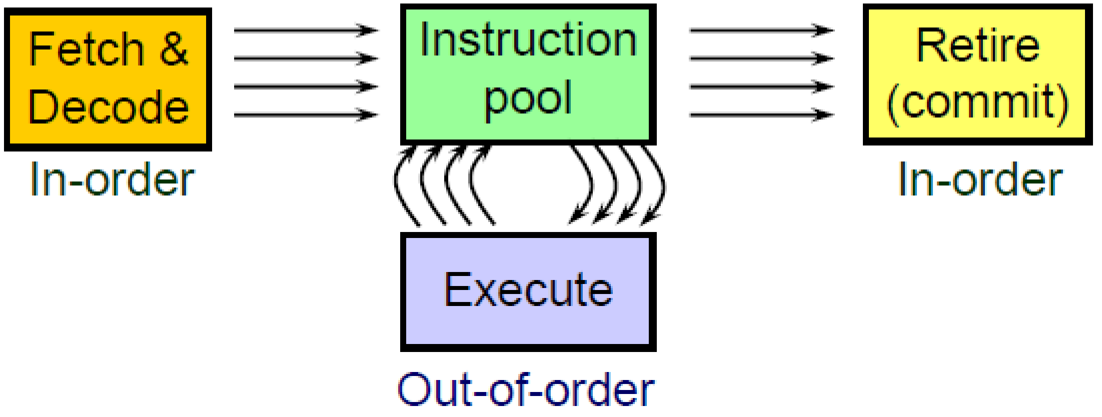
In-order execution



Out-of-order execution



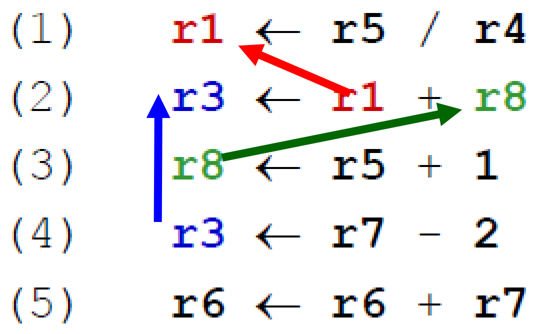
OOOE – General Scheme



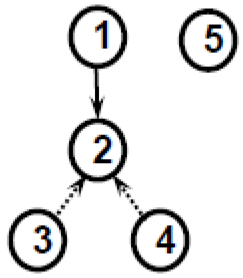
- Fetch & decode instructions in parallel but in order, to fill instruction pool
- Execute ready instructions from the instructions pool
  - All the data required for the instruction is ready
  - Execution resources are available
- Once an instruction is executed
  - signal all dependant instructions that data is ready
- Commit instructions in parallel but in-order
  - Can commit an instruction only after all preceding instructions (in program order) have committed

Example

- Assume that executing a divide operation takes 20 cycles



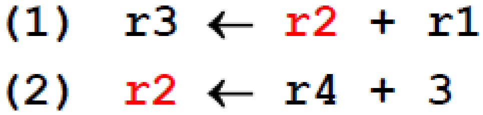
```
(1)   r1 ← r5 / r4
(2)   r3 ← r1 + r8
(3)   r8 ← r5 + 1
(4)   r3 ← r7 - 2
(5)   r6 ← r6 + r7
```



- Inst2 has a **RAW** dependency on r1 with Inst1
  - It cannot be executed in parallel with Inst1
- Can successive instructions pass Inst2 ?
  - Inst3 cannot since Inst2 must read r8 before Inst3 writes to it
  - Inst4 cannot since it must write to r3 after Inst2
  - Inst5 can

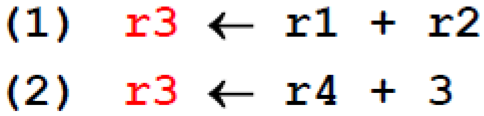
False Dependencies

- OOOE creates new dependencies
  - **WAR**: write to a register which is read by an earlier inst.



```
(1)  r3 ← r2 + r1
(2)  r2 ← r4 + 3
```

  - **WAW**: write to a register which is written by an earlier inst.



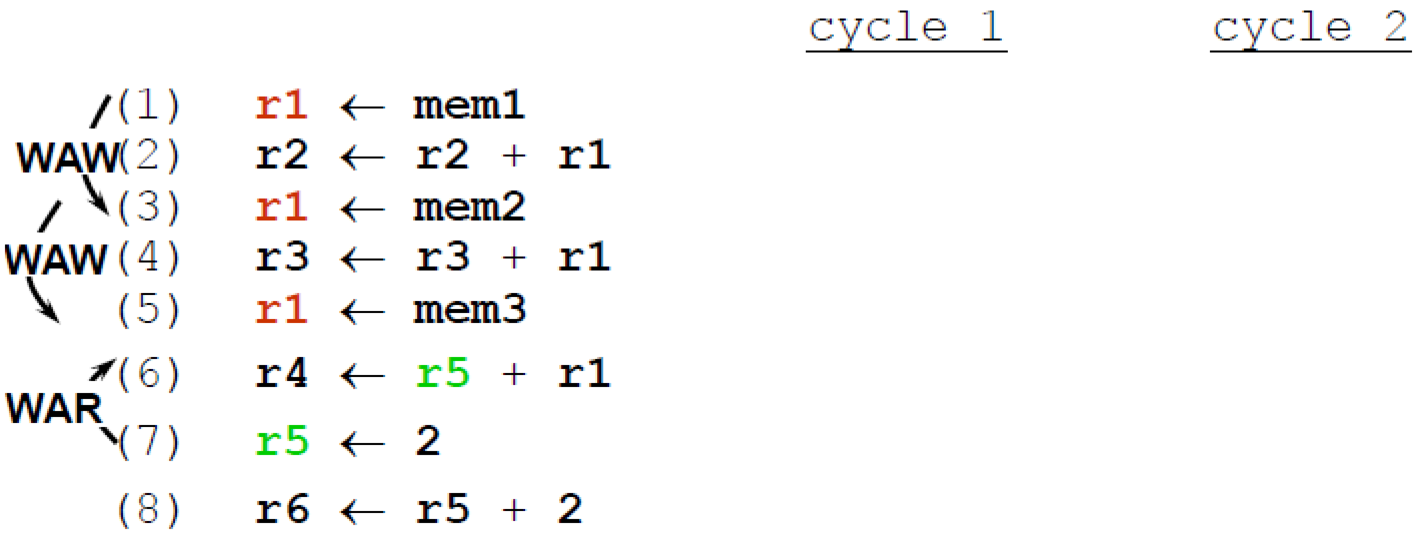
```
(1)  r3 ← r1 + r2
(2)  r3 ← r4 + 3
```

- These are *false dependencies*
  - There is no missing data
  - Still prevent executing instructions out-of-order
- Solution: *Register Renaming*

Register Renaming

- Hold a pool of *physical registers*
- Map architectural registers into physical registers
  - Before an instruction can be sent for execution
    - **Allocate** a free physical register from a pool
    - The physical register **points** to the architectural register
  - When an instruction **writes** a result
    - Write the result value to the **physical register**
  - When an instruction needs data from a register
    - **Read** data from the **physical register** allocated to the latest inst which writes to the same arch register, and precedes the current instruction
    - If no such instruction exists, read directly from the arch register
  - When an instruction **commits**
    - **Move** the value from the **physical register** to the **arch register** it points

Example



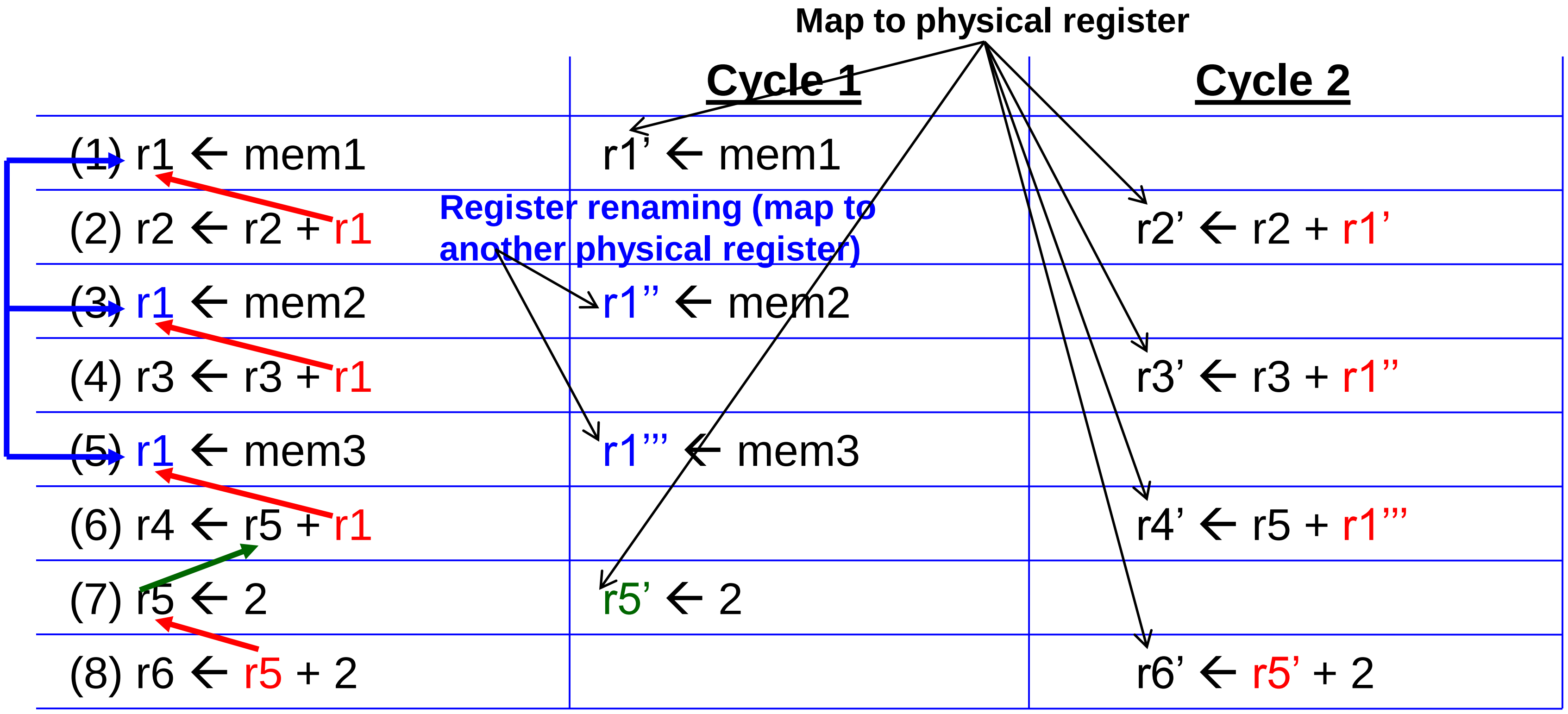
cycle 1 cycle 2

```
(1)   r1 ← mem1
(2)   r2 ← r2 + r1
(3)   r1 ← mem2
(4)   r3 ← r3 + r1
(5)   r1 ← mem3
(6)   r4 ← r5 + r1
(7)   r5 ← 2
(8)   r6 ← r5 + 2
```

Register Renaming Benefits

- Removes false dependencies
- Removes architecture limit for # of registers

Mapping to Physical Registers



Map to physical register

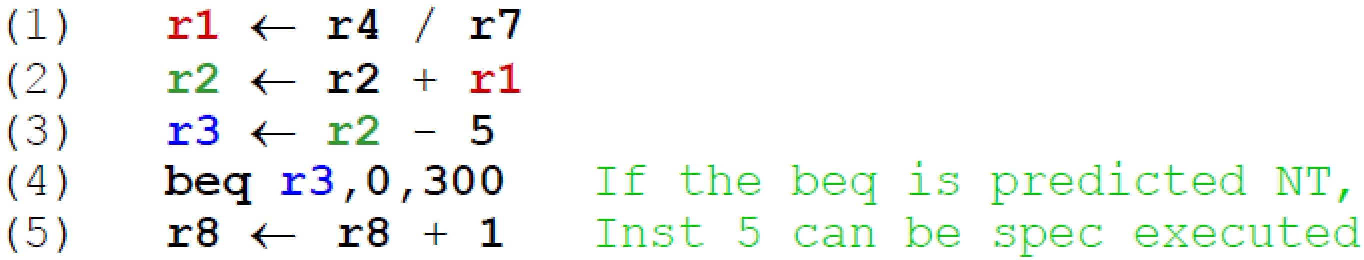
Register renaming (map to another physical register)

	Cycle 1	Cycle 2
(1) r1 ← mem1	r1’ ← mem1	
(2) r2 ← r2 + r1		r2’ ← r2 + r1’
(3) r1 ← mem2	r1” ← mem2	
(4) r3 ← r3 + r1		r3’ ← r3 + r1”
(5) r1 ← mem3	r1’” ← mem3	
(6) r4 ← r5 + r1		r4’ ← r5 + r1’”
(7) r5 ← 2	r5’ ← 2	
(8) r6 ← r5 + 2		r6’ ← r5’ + 2

- r1, r2, … , r6 is referring to the architectural register while r1', r1" etc. are referring to physical registers. There can be multiple of them, depending on the processor.
- The results from each instruction will only be committed to the architectural registers in the in-order retirement stage.

Executing Beyond Branches

- So far we do not look for instructions ready to execute beyond a branch
- Limited to the parallelism within a basic-block
  - A basic-block is ~5 instruction long



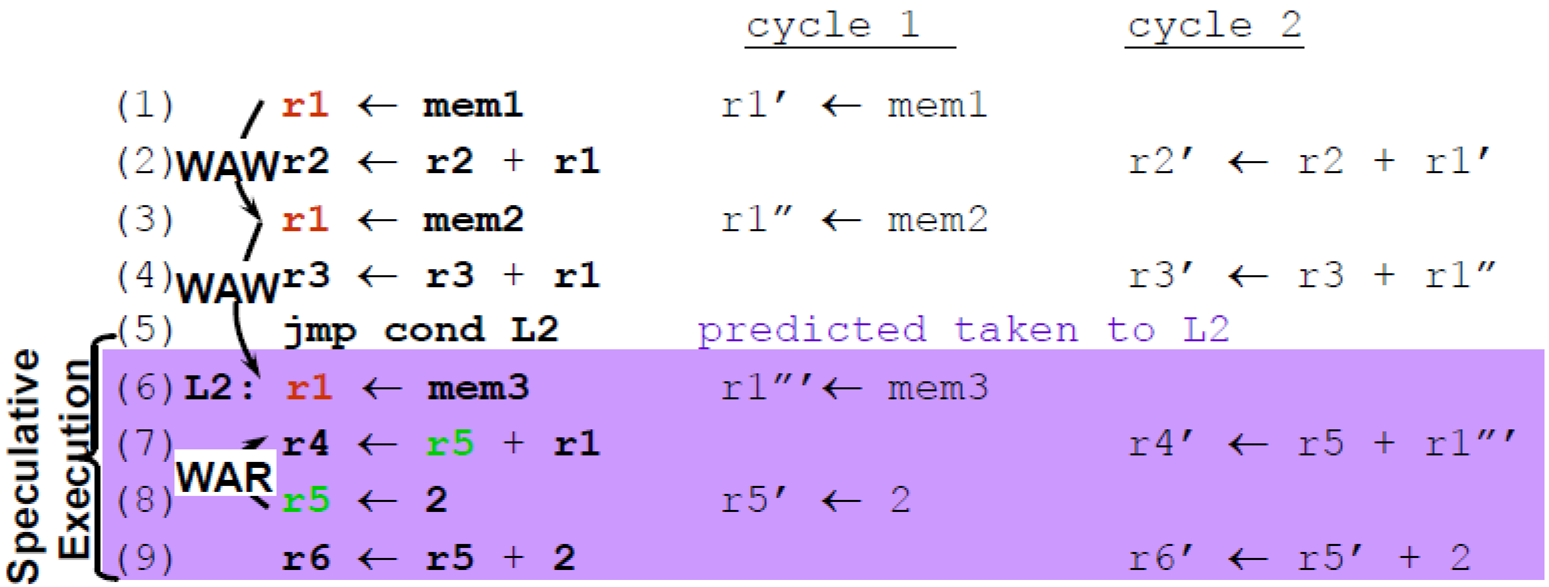
```
(1)    r1 ← r4 / r7
(2)    r2 ← r2 + r1
(3)    r3 ← r2 - 5
(4)    beq r3,0,300    If the beq is predicted NT,
(5)    r8 ←  r8 + 1    Inst 5 can be spec executed
```

- We would like to look beyond branches
  - But what if we execute an instruction beyond a branch and then it turns out that we predicted the wrong path ?
- Solution: *Speculative Execution*

Speculative Execution

- Execution of instructions from a predicted (yet unsure) path
  - Eventually, path may turn wrong
- Implementation:
  - Hold a pool of all *not yet executed instructions*
    - Fetch instructions into the pool from a predicted path
  - Instructions for which all operands are *ready can be executed*
  - An instruction may change the processor state (commit) only when it is safe
    - An instruction commits only when all previous (in-order) instructions have committed instructions commit in-order
    - Instructions which follow a branch commit only after the branch commits
      - If a predicted branch is wrong all the instructions which follow it are flushed
- Register Renaming helps speculative execution
  - Renamed registers are **kept** until speculation is verified to be correct

Example



		cycle 1	cycle 2
(1)	r1 ← mem1	r1′ ← mem1	
(2)	r2 ← r2 + r1		r2′ ← r2 + r1′
(3)	r1 ← mem2	r1″ ← mem2	
(4)	r3 ← r3 + r1		r3′ ← r3 + r1″
(5)	jmp cond L2	predicted taken to L2	
(6)	L2: r1 ← mem3	r1‴← mem3	
(7)	r4 ← r5 + r1		r4′ ← r5 + r1‴
(8)	r5 ← 2	r5′ ← 2	
(9)	r6 ← r5 + 2		r6′ ← r5′ + 2

- Instructions 6-9 are speculatively executed
  - If the prediction turns wrong, they will be flushed
- If the branch was predicted taken
  - The instructions from the other path would be have been speculatively executed

Data Dependencies in Loops

- Not all data dependencies are obvious by examining the source code.
- Consider

```
 do I = 2, n
        X(I) = A*X(I-1) + B
```

- Recurrence between iteration I and I-1
- Compiler needs to analyze loop expressions to detect and handle recurrences correctly.

Control Dependencies

- Consider the following code

```
        mul      r1, r2 → r3
        jz       zproc
        sub r4, r1 → r1
        :
zproc:  load r1, x
        :
```

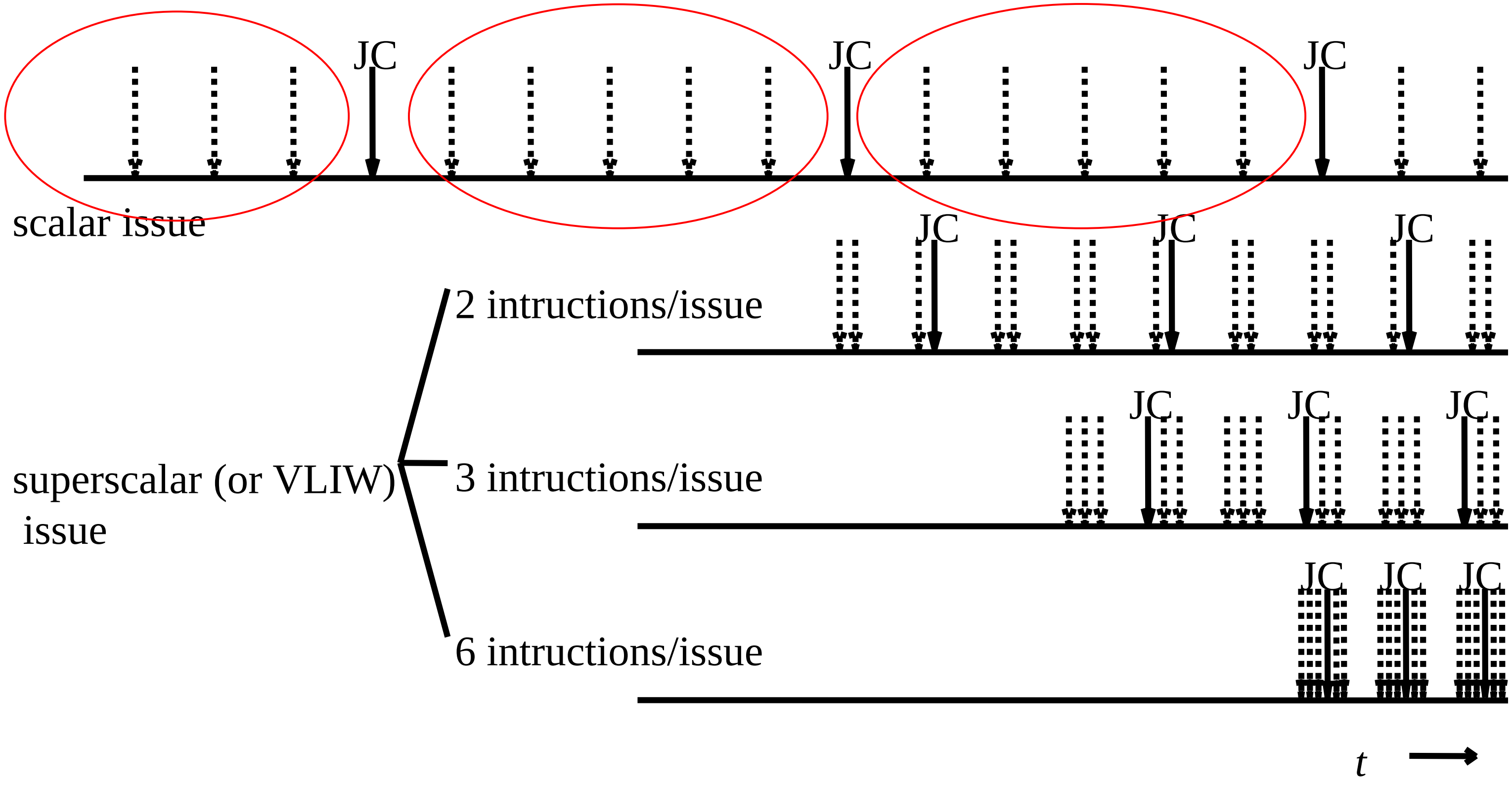
Branch depends on
Result of “mul”

What happens if we
DON’T
Prefetch this instruction?

Is this a problem?

- Ratio of branches
  - General Purpose 22 % to 39%
  - Scientific 5% - 11 %
- Conditional to unconditional
  - General Purpose 46% to 83%
  - Scientific 53 % to 83 %
- For 25% 1 in 4 instruction is a branch
- Effect on pipeline?

Impact on ILP processors



JC: conditional branch

Resource Dependencies

- Consider the following code

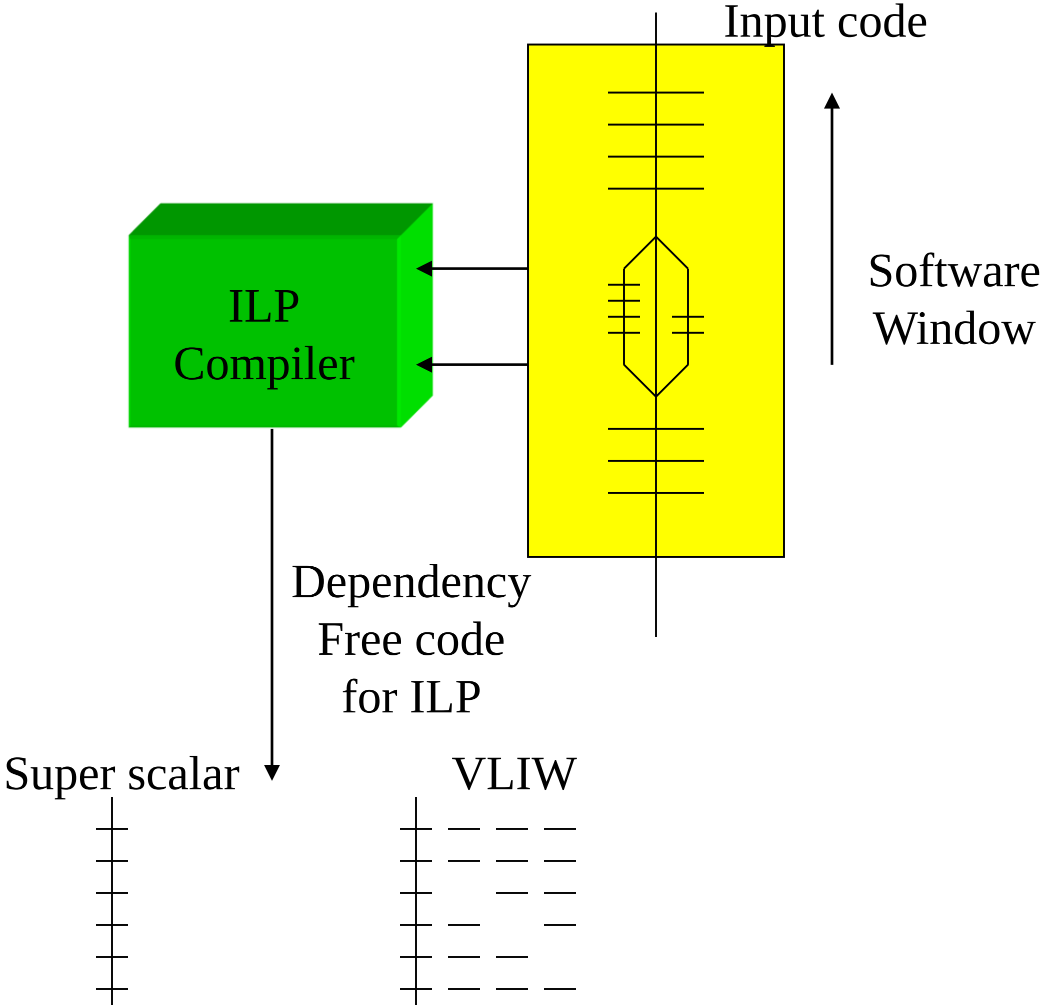
```
div        r1, r2 → r3
div        r4, r2 → r5
```

- Insufficient dividers to issue both instructions in parallel
- Possible resources
  - buses, execution units, buffers (storage slots)

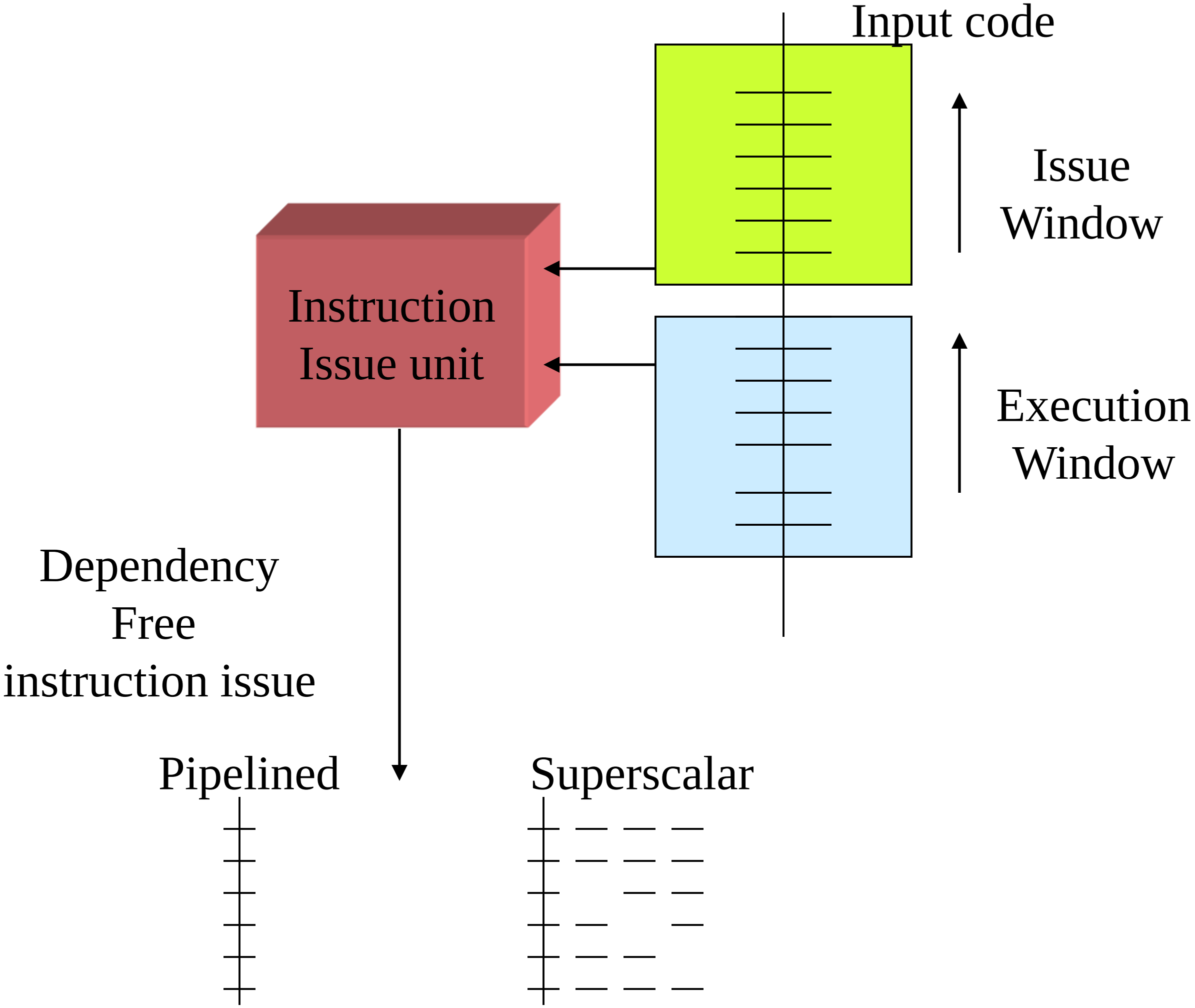
Instruction scheduling

- When instructions are processed (that is, issued, renamed, executed and so forth) in parallel, it is often necessary to detect and resolve dependencies between instructions.
- Three main approaches
  - Static scheduling
  - Dynamic scheduling
  - Hybrid

Static vs Dynamic scheduling



Static vs Dynamic scheduling ...



ILP Instruction Scheduling

- Static Scheduling boosted by parallel code optimisation
  - Performed entirely by the compiler
  - Processor receives dependency-free and optimized code for parallel execution
  - *Typical for VLIW and a few pipelined processors (e.g. MIPS)*

ILP Instruction Scheduling

- Dynamic Scheduling without static parallel code optimisation
  - Performed entirely by the processor
  - The code is not optimised for parallel execution. The Processor detects and resolves dependencies on its own
  - *Early ILP processors (e.g. CDC 6600, IBM 360/91)*

ILP Instruction Scheduling

- Dynamic scheduling boosted by static parallel code optimisation
  - Performed by the processor in conjunction with the parallel optimizing compiler
  - The processor receives optimised code for parallel execution, but it detects and resolves dependencies of its own
  - *Usual practice for pipelined superscalar processors (e.g. RS/6000, PL.8/XL, i960 and QTC SF960)*

Preserving Sequential Consistency

- When instructions are executed in parallel, processor must be careful to preserve the sequential consistency.

  div r1, r2 → r3

  add r5, r6 → r7

  jz anywhere

- Must make sure the *jz* instruction uses the condition codes set by the *add*, not the *div* which might take longer.

Types of consistency

- Strong consistency
  - Preserves the actual execution order
- Weak consistency
  - Produces correct result
  - Can execute out of order providing the code still delivers the correct result

How fast can we go?

- Performance is limited by
  - underlying algorithm (dependencies)
  - compiled code (false dependencies, staging)
  - actual hardware (resource restrictions)
- Study all three to work out maximum speedup

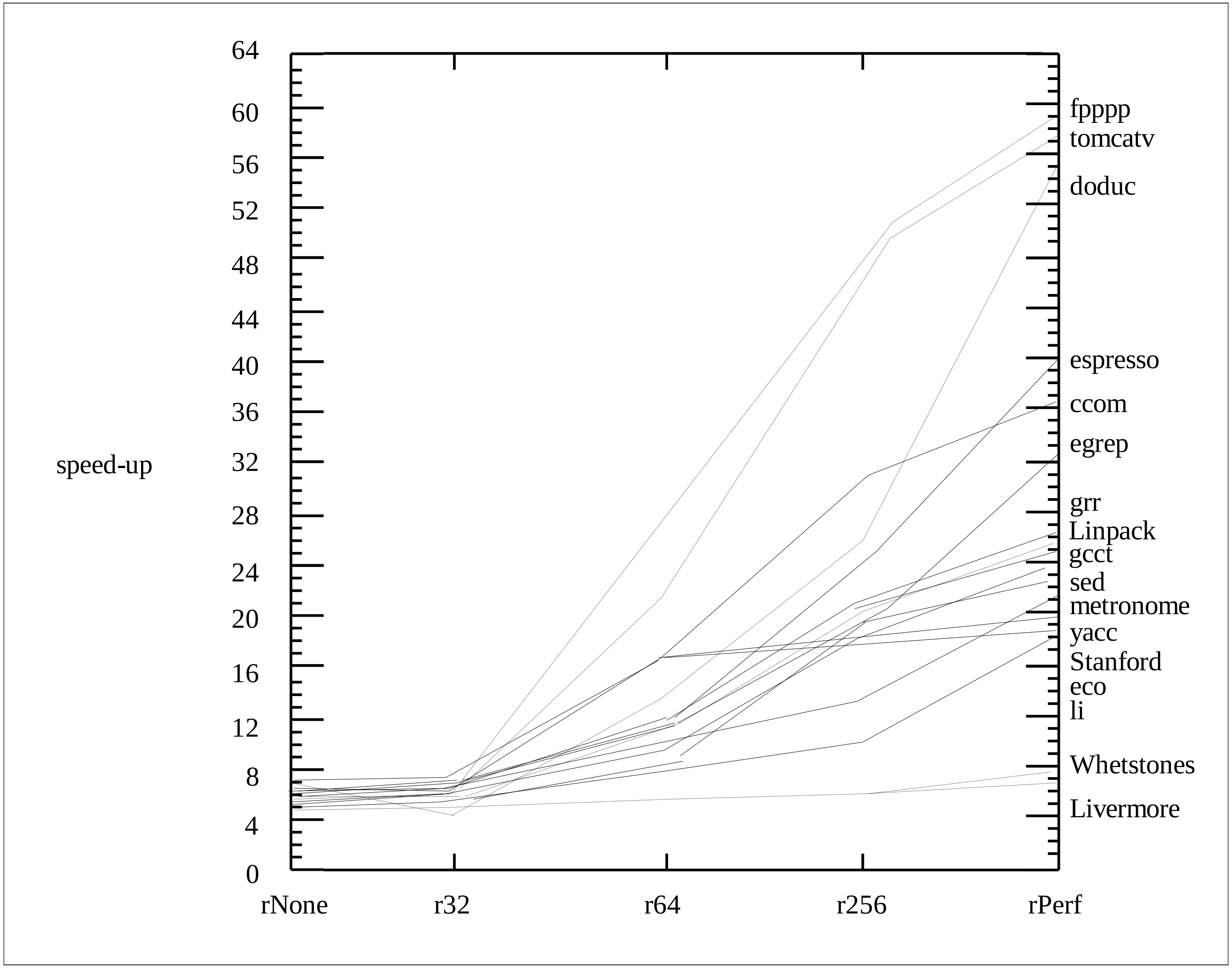
How fast can we go?
Earliest methods to achieve speed ups for scientific programs

Paper	Benchmark	Speed-up for general programs		Speed-up for scientific programs	
		range	average	range	average
Tjaden, Flynn 1970	31 Library programs			1.2 - 3.2	**1.9**
Kuck & at. 1972	20 FORTRAN programs			1.2 - 17	**### 4**
Riseman, Foster 1972	7 FORTRAN/ assembly programs	1.2 - 3.0	**1.8**	1.4 - 1.6	**1.6**
Jouppy 1989	8 Modula-2 programs	1.6 - 2.2	**1.9**	2.4 - 3.3	**2.8**
Lam, Wilson 1992	6 SPECmarks + 4 others	1.5 - 2.8	**2.1**	2 - 293	

How fast can we go?
Earliest methods to achieve speed ups for scientific programs

Paper	Benchmark	Speed-up for general purpose programs		Speed-up for scientific programs	
		range	average	range	average
Riseman, Foster 1972	7 FORTRAN/ assembly programs	8 - 100	**42**	30 - 120	**75**
Nicolaus, Fisher 1984	22 numerical programs			3 - 988	**12**
Kumar 1988	4 FORTRAN programs			475 - 3500	**1839**
Butler & at.1991	9 SPEC benchmarks	38 - 200	**170**	17 - 1165	**509**
Wall 1991	6SPECbenchmarks, 13 others	16 -41 7 - 37	**28** **24**	57 - 60 6 - 27	**59** **18**
Austin, Sohi 1992	SPEC benchmarks	13 - 942	**288**	51 - 33749	**6188**
Lam, Wilson 1992	6SPEC benchmarks, 4 others	174 - 3283 47 - 265	**1400** **229**	844 - 188 470	**64500**

Effect of register renaming



Summary

- Evolution and overview of ILP Processors
  - Pipelines, VLIW or super scaler
- Understanding of inter-instruction dependencies
  - Data, Control and Resource Dependencies
- Instruction Scheduling
  - Static & Dynamic scheduling
- Sequential Consistency
  - When instructions are executed in parallel, processor must be careful to preserve the sequential consistency.
- How fast can we go?
  - Performance is limited by underlying algorithm compiled code and actual hardware (resource restrictions)

References

Sima .D, Fountain .T, Karsuk .P, “Chapter 4: Introduction to ILP-Processors”, in *Advanced Computer Architectures – A Design Space Approach*, Addison Wesley Longman, 1997, pp. 113 – 135.