COMPUTER ARCHITECTURES

Out-of-order Execution

Budapest,
4/10/25

Gábor Horváth, ghorvath@hit.bme.hu

EXAMPLE (REMINDER)

C code:

```
for (i=0; i<N; i++)
    Z[i]=A*X[i];
```

Elementary instructions:

```
D2 ← MEM[R1]
D3 ← D2 * D0
MEM[R2] ← D3
R1 ← R1 + 8
R2 ← R2 + 8
```

A: D0
X[i]: MEM[R1]
Z[i]: MEM[R2]

Instruction scheduling: (latency of multiplication: 5, integer and memory operations: 1)

Instructions:	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.
D2 ← MEM[R1]	IF	ID	EX	MEM	WB									
D3 ← D2 * D0		IF	ID	**D***	M0	M1	M2	M3	M4	MEM	WB			
MEM[R2] ← D3			IF	**S***	ID	**D***	**D***	**D***	**D***	EX	MEM	WB		
R1 ← R1 + 8					IF	**S***	**S***	**S***	**S***	ID	EX	MEM	WB	
R2 ← R2 + 8										IF	ID	EX	MEM	WB

OPTIMIZATION

- Let us determine the optimal order of instructions

Original:

```
D2 ← MEM[R1]
D3 ← D2 * D0
MEM[R2] ← D3
R1 ← R1 + 8
R2 ← R2 + 8
```


Optimized:

```
D2 ← MEM[R1]
D3 ← D2 * D0
R1 ← R1 + 8
MEM[R2] ← D3
R2 ← R2 + 8
```

Instructions:	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.
D2 ← MEM[R1]	IF	ID	EX	MEM	WB									
D3 ← D2 * D0		IF	ID	**D***	M0	M1	M2	M3	M4	MEM	WB			
R1 ← R1 + 8			IF	**S***	ID	EX	MEM	WB						
MEM[R2] ← D3					IF	ID	**D***	**D***	**D***	EX	MEM	WB		
R2 ← R2 + 8						IF	**S***	**S***	**S***	ID	EX	MEM	WB	

OPTIMIZATION

- The program become faster due to the optimization
- Problems of this approach:
  - Programmers/compiler have to recognize these possibilities
  - The optimal order depends on the structure of the pipeline
- Ideal solution:
  - The CPU should do it, on-the-fly
  - The CPU should re-order the instructions such that
    - the execution time is lower
    - the semantics of the program remains the same

  → **Out-of-order execution**
- Implementation:
  - Not as complex as it seems
  - In use for a long time
    - First: Scoreboard – 1964
    - Advanced: **Tomasulo** – 1967
      - Used even in recent CPUs (e.g. Intel Core i7)

Out-of-order execution

THE FIRST

- First out-of-order CPU
  - 1964: CDC 6600
- Designed by Seymour Cray
- Features:
  - 16 functional units
  - 10 MHz clock rate
  - >400.000 transistors
  - 5 tons
  - Wired control
- The fastest computer of the word for the next 5 years

CDC-6600

Freon-based cooling in each case:

- Dual vector-graphics display:

CRAY VS. IBM

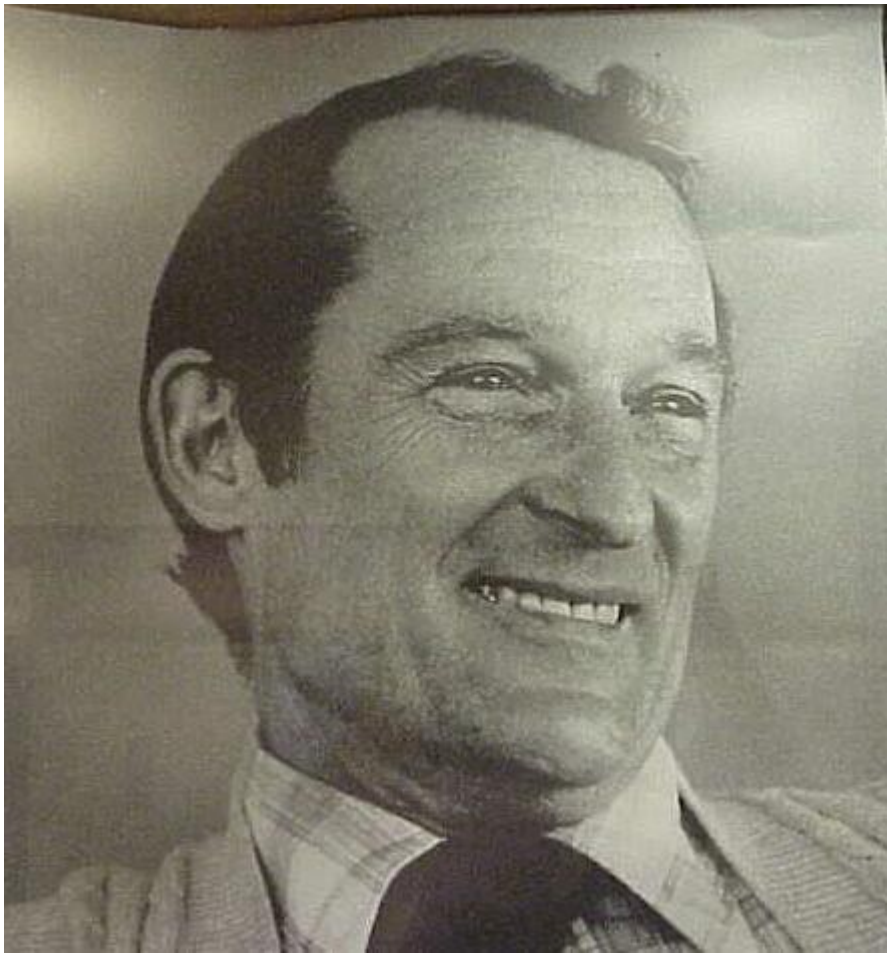
- **Thomas Watson (IBM):**

  „Last week, Control Data ... announced the 6600 system. I understand that in the laboratory developing the system there are only 34 people including the janitor. Of these, 14 are engineers and 4 are programmers... Contrasting this modest effort with our vast development activities, I fail to understand why we have lost our industry leadership position by letting someone else offer the world's most powerful computer.”

CRAY VS. IBM

- **Seymour Cray (CD)**

  „It seems like Mr. Watson has answered his own question.”



THE INGREDIENTS OF OUT-OF-ORDER EXECUTION

- The CPU has to fetch several instructions to be able to re-order them
  - They must be stored somewhere:

    → **instruction window (instruction pool, reservation station)**
- A procedure that determines the execution order of the instructions

  → **dynamic scheduler**
- An idea to reduce the number of dependencies between the instructions

  → **register renaming**
- A trick to show in-order execution to the outside world

  → **re-order buffer**

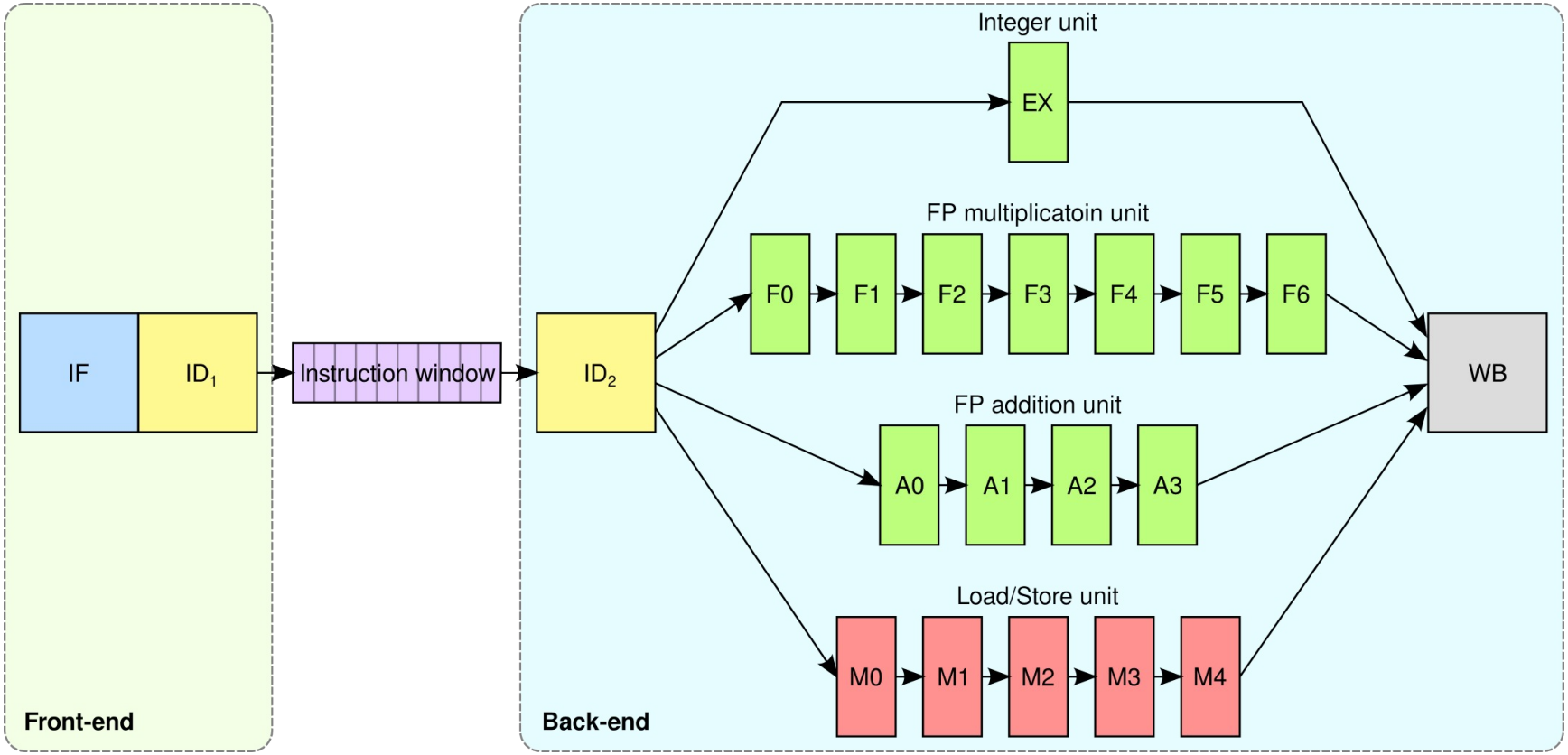
THE INGREDIENTS OF OUT-OF-ORDER EXECUTION

- The CPU has to fetch several instructions to be able to re-order them
  - They must be stored somewhere:
    → **instruction window (instruction pool, reservation station)**
- A procedure that determines the execution order of the instructions
  → **dynamic scheduler**
- An idea to reduce the number of dependencies between the instructions
  → **register renaming**
- A trick to show in-order execution to the outside world
  → **re-order buffer**

INSTRUCTION WINDOW / RESERVATION STATIONS

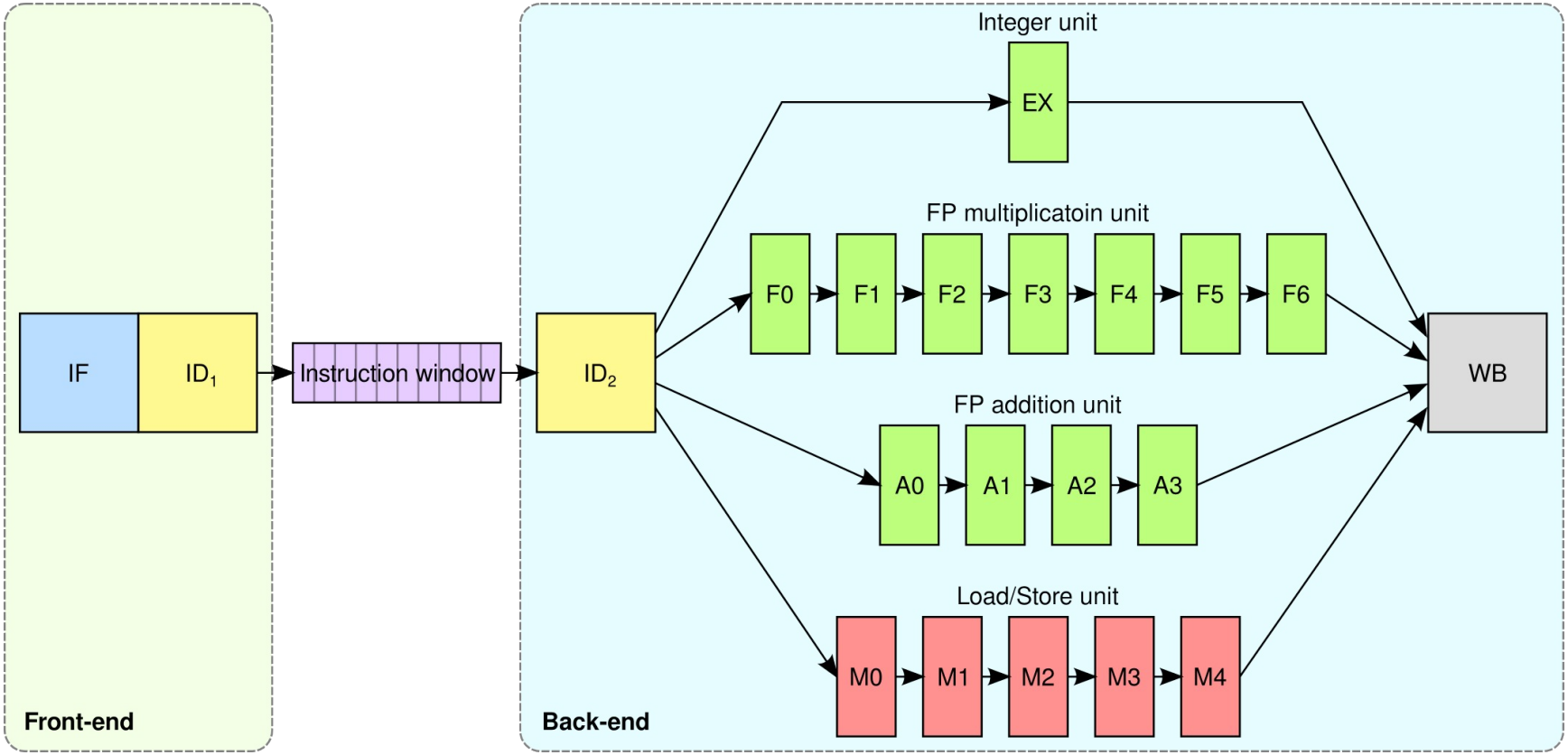
- Fetching instructions (IF) → in-order
- Executing instructions (EX) → out-of-order

  → the instruction window is located in the ID phase



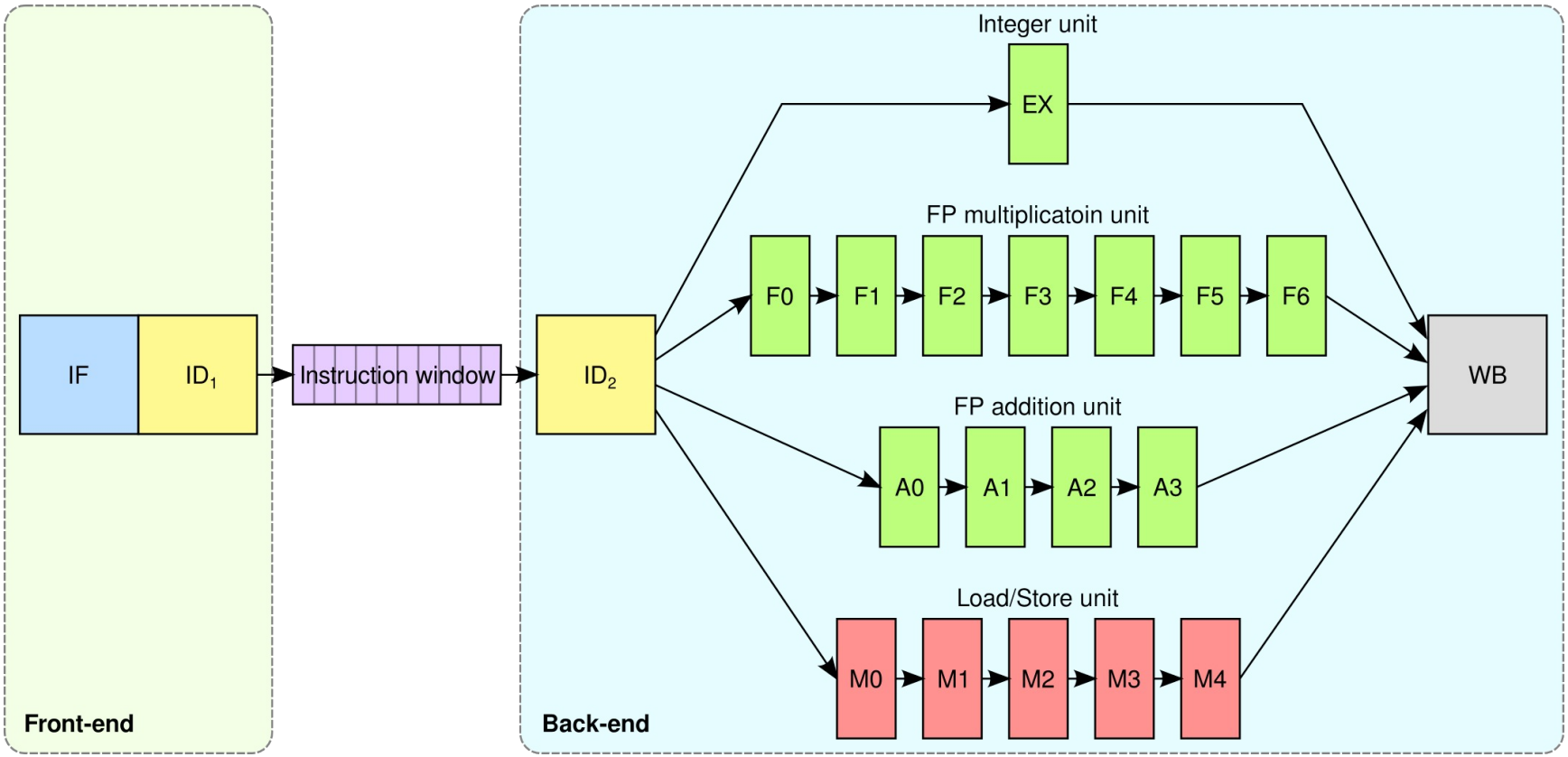
INSTRUCTION WINDOW / RESERVATION STATIONS

- Cuts the pipeline into two parts:
  - Front-end: instructions enter in an in-order way
  - Back-end: instructions enter in an out-of-order way



INSTRUCTION WINDOW / RESERVATION STATIONS

- ID is now separated to (non-standard terminology!):
  - ID1: Dispatch (DS) – decodes instructions and puts them into the instruction window
  - ID2: Issue (IS) – collects operands and assigns instructions to functional units



THE INGREDIENTS OF OUT-OF-ORDER EXECUTION

- The CPU has to fetch several instructions to be able to re-order them
  - They must be stored somewhere:
    → **instruction window (instruction pool, reservation station)**
- A procedure that determines the execution order of the instructions
  → **dynamic scheduler**
- An idea to reduce the number of dependencies between the instructions
  → **register renaming**
- A trick to show in-order execution to the outside world
  → **re-order buffer**

THE INGREDIENTS OF OUT-OF-ORDER EXECUTION

- The CPU has to fetch several instructions to be able to re-order them
  - They must be stored somewhere:
    → **instruction window (instruction pool, reservation station)**
- A procedure that determines the execution order of the instructions
  → **dynamic scheduler**
- An idea to reduce the number of dependencies between the instructions
  → **register renaming**
- A trick to show in-order execution to the outside world
  → **re-order buffer**

DYNAMIC INSTRUCTION SCHEDULING

- The execution order of instructions is determined using

  Data-flow approach

  - A precedence graph is created
    - Nodes: instructions
    - Arcs: which other instructions need to be awaited before executing the instruction → dependencies
  - An instruction is ready to be executed *if all its dependencies are resolved*. The dependencies can be:
    - RAW: an input parameter is not yet ready
    - WAW: it would write to a register, which is still to be written by an earlier, not yet finished instruction
    - WAR: it would write to a register, which is still to be read by an earlier, not yet finished instruction
  - If an instruction is ready to be executed, and there is an appropriate execution unit available, we execute it immediately

DYNAMIC INSTRUCTION SCHEDULING

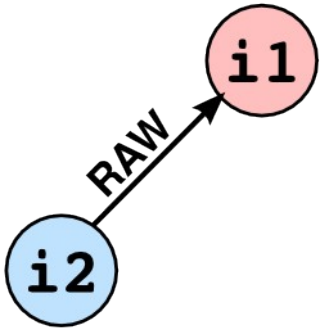
- What happens, if more than one instructions are ready for an execution unit? (Contention)
  - Choose the one that was ready earlier
  - Choose the one that prevents the highest number of instruction from being ready for execution
  - etc.

- For coloring the nodes of the precedence graph, we use the following colors to express the status of the instructions
  - Blue: instruction is loaded into the reservation station
  - Red: the execution of the instruction is in progress
  - Green: the execution of the instruction has been finished

EXAMPLE

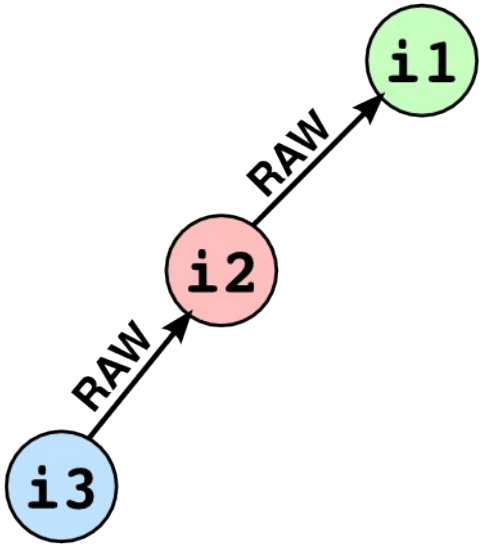

```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

EXAMPLE



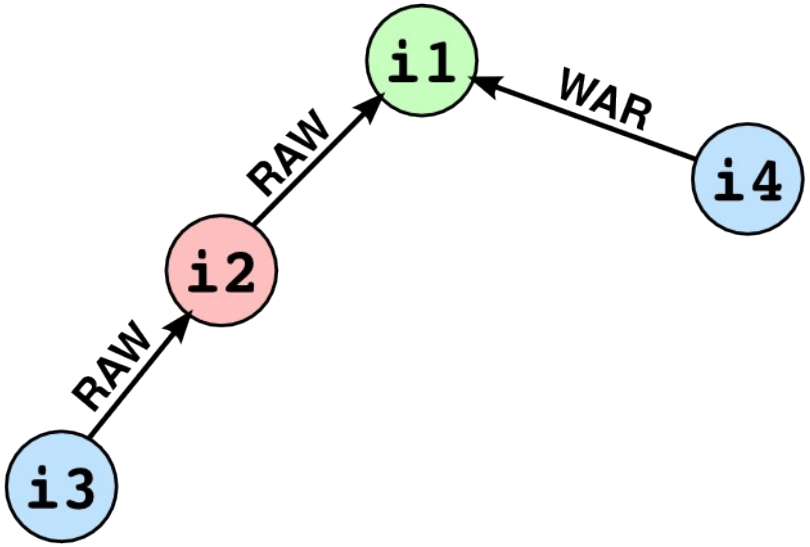

```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

EXAMPLE



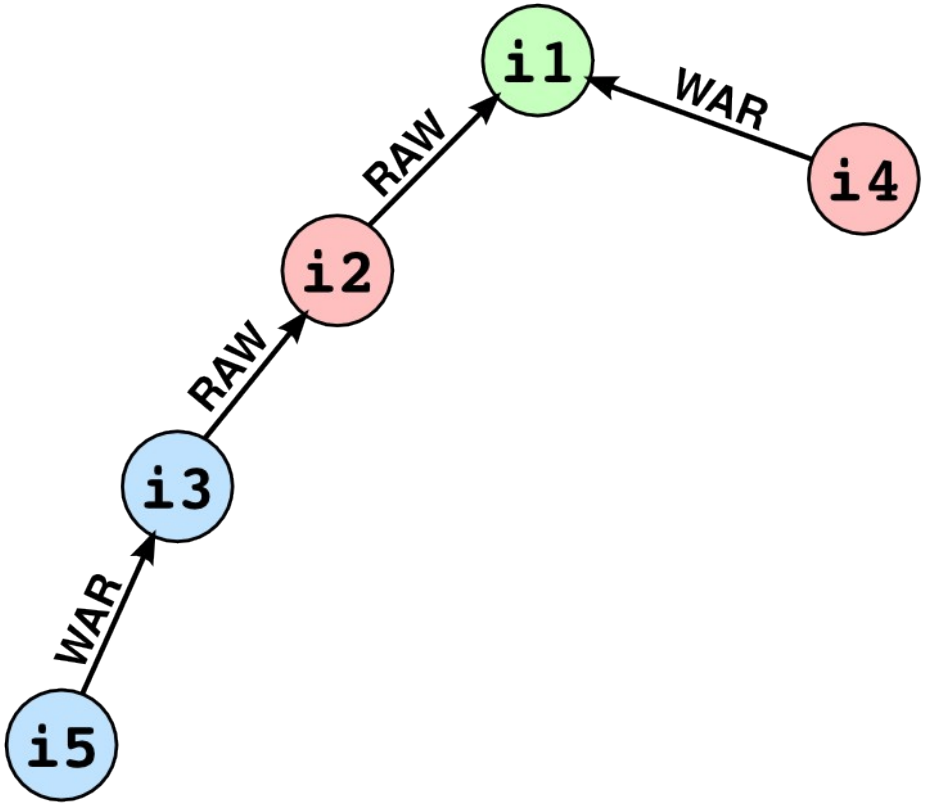

```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

EXAMPLE



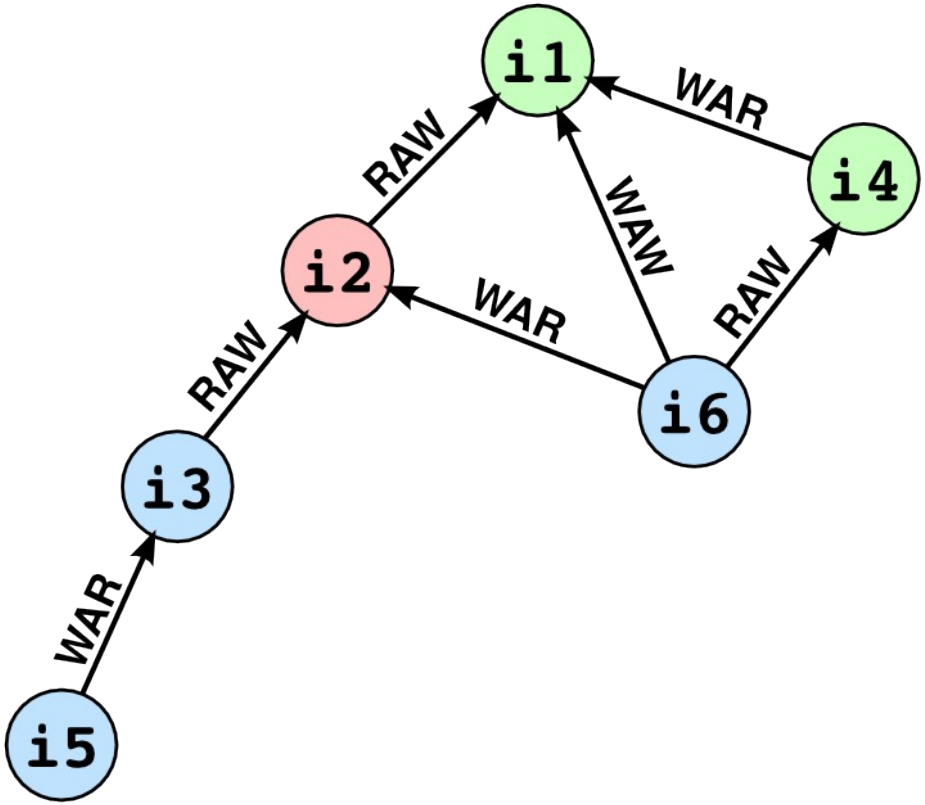

```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

EXAMPLE



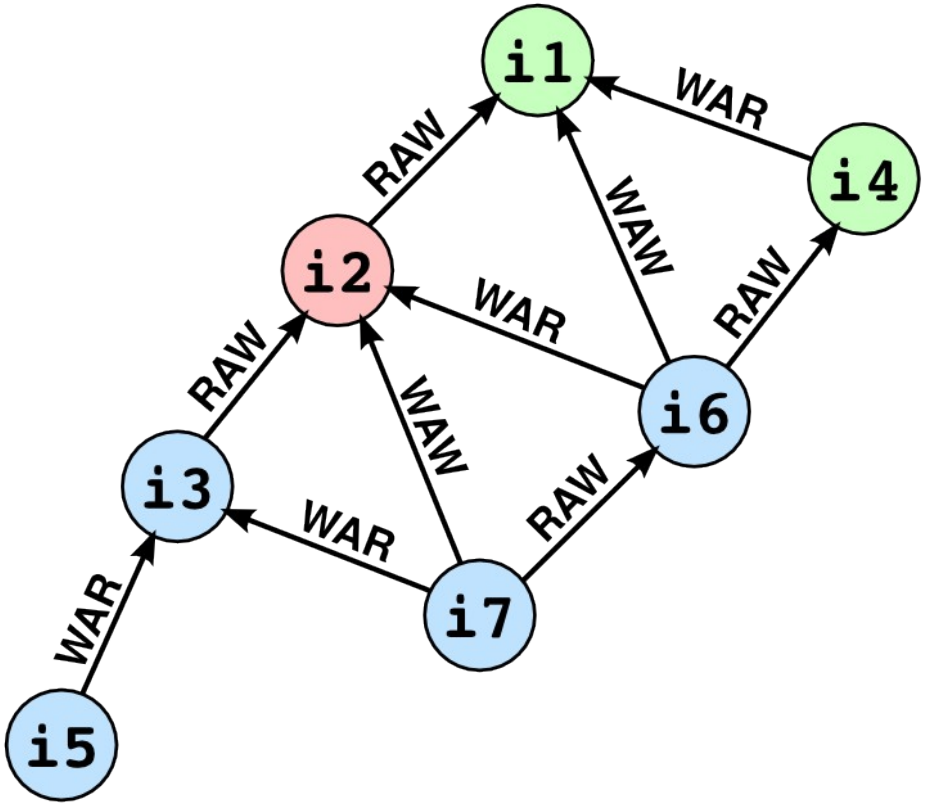

```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

EXAMPLE



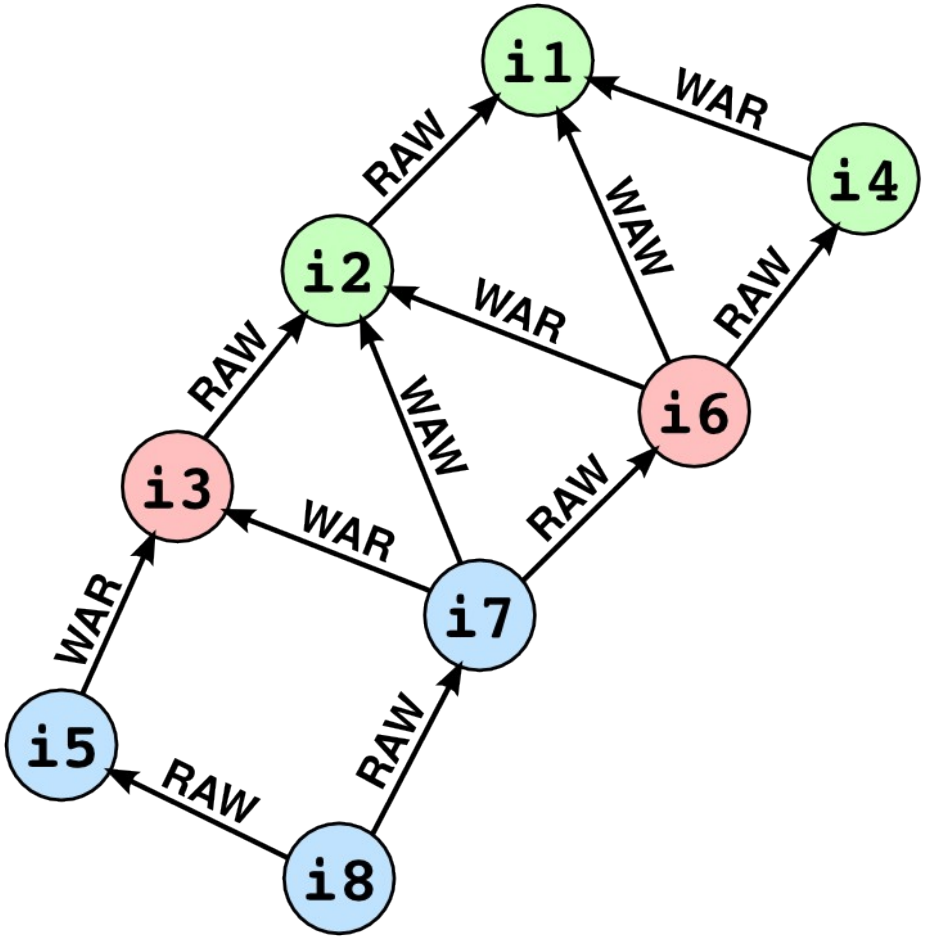

```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

EXAMPLE



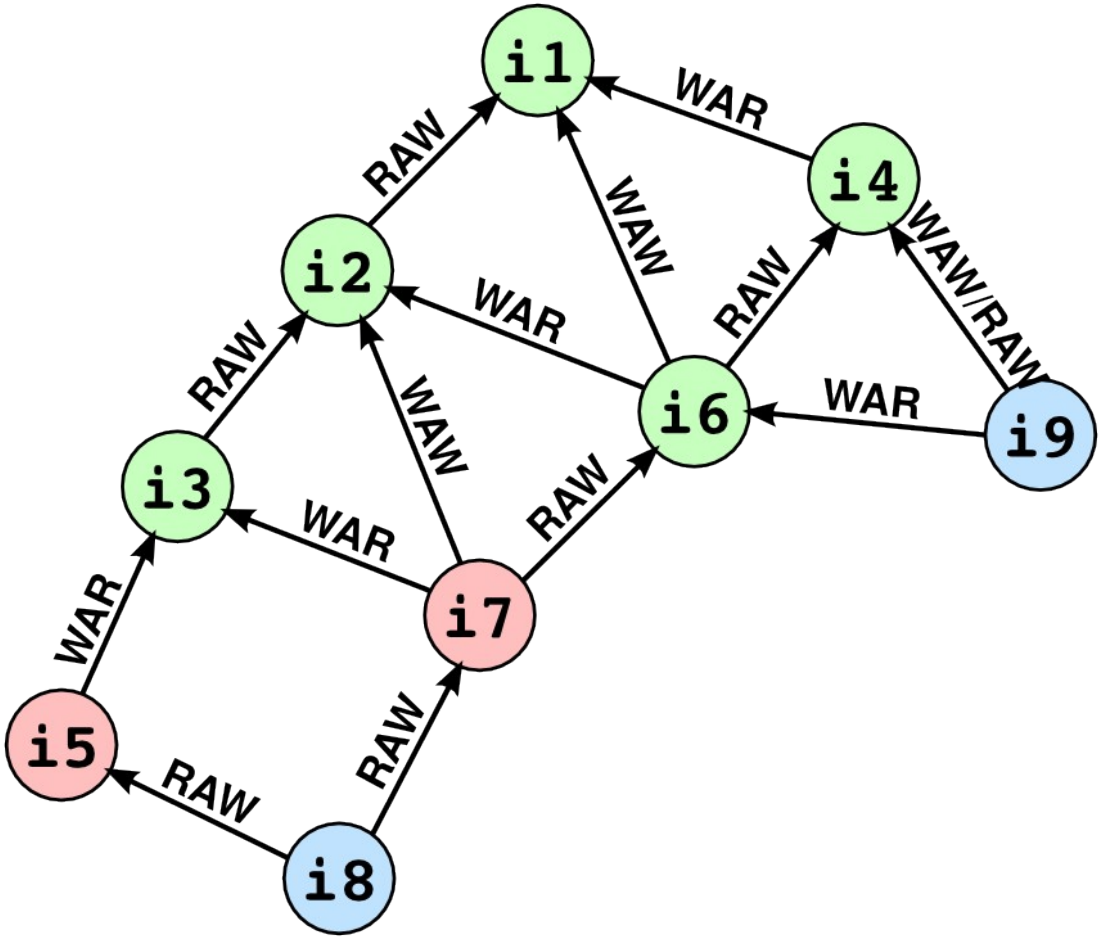

```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

EXAMPLE




```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

EXAMPLE




```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

THE INGREDIENTS OF OUT-OF-ORDER EXECUTION

- The CPU has to fetch several instructions to be able to re-order them
  - They must be stored somewhere:
    → **instruction window (instruction pool, reservation station)**
- A procedure that determines the execution order of the instructions
  → **dynamic scheduler**
- An idea to reduce the number of dependencies between the instructions
  → **register renaming**
- A trick to show in-order execution to the outside world
  → **re-order buffer**

THE INGREDIENTS OF OUT-OF-ORDER EXECUTION

- The CPU has to fetch several instructions to be able to re-order them
  - They must be stored somewhere:

    → **instruction window (instruction pool, reservation station)**
- A procedure that determines the execution order of the instructions

  → **dynamic scheduler**
- An idea to reduce the number of dependencies between the instructions

  → **register renaming**
- A trick to show in-order execution to the outside world

  → **re-order buffer**

REGISTER RENAMING

- Goal: to make the precedence graph sparse
- The types of data dependencies (hazards):
  - RAW:

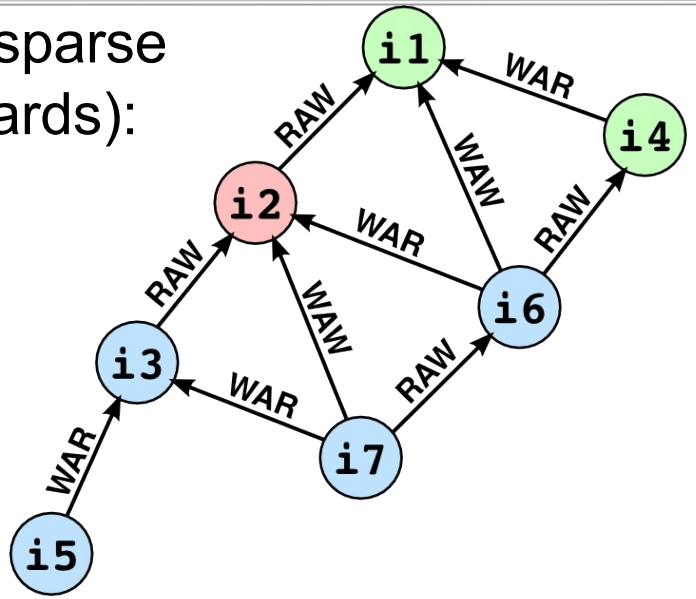
```
D3 ← D2 * D0
MEM[R2] ← D3
```

    → Real dependency
  - WAR:

```
D3 ← D2 * D0
...
D2 ← MEM[R1]
```

  - WAW:

```
D3 ← D2 * D0
...
D3 ← MEM[R1]
```



- Surprise: the WAR and the WAW dependencies can be eliminated
  → Anti-dependencies

REGISTER RENAMING

- Why would a programmer write such a code?
  - WAR:

```
D3 ← D2 * D0
...
D2 ← MEM[R1]
```

  - WAW:

```
D3 ← D2 * D0
...
D3 ← MEM[R1]
```

- The registers are re-used because we have too few of them!
- Let us put a large number of registers to the CPU
  → **Physical registers**
- Let us hide them. The programmer still works with
  → **Logical/architectural registers (described in the ISA)**
- The CPU re-writes the program on the fly to make use of the physical registers
  → **Register renaming**

REGISTER RENAMING

- Tool: **Register alias table**
  - Entry i.: the ID of the physical register assigned to logical register i
- Procedure:
  - The operands are replaced by physical registers
  - The result is put to a brand new (unused) physical register
  - The register alias table is updated

Original:

```
i1:  D2 ← MEM[R1]
```

After renaming:

```
i1:  U25 ← MEM[T3]
```

Register alias table:

Logical reg.	Physical reg.
R0	T21
R1	T3
R2	T46
R3	T8
D0	U9
D1	U24
D2	U17
D3	U4

REGISTER RENAMING

- Tool: **Register alias table**
  - Entry i.: the ID of the physical register assigned to logical register i
- Procedure:
  - The operands are replaced by physical registers
  - The result is put to a brand new (unused) physical register
  - The register alias table is updated

Original:

```
i1:  D2 ← MEM[R1]
```

After renaming:

```
i1:  U25 ← MEM[T3]
```

Register alias table:

Logical reg.	Physical reg.
R0	T21
R1	T3
R2	T46
R3	T8
D0	U9
D1	U24
D2	**U25**
D3	U4

REGISTER RENAMING

- Tool: **Register alias table**
  - Entry i.: the ID of the physical register assigned to logical register i
- Procedure:
  - The operands are replaced by physical registers
  - The result is put to a brand new (unused) physical register
  - The register alias table is updated

Original:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
```

After renaming:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
```

Register alias table:

Logical reg.	Physical reg.
R0	T21
R1	T3
R2	T46
R3	T8
D0	U9
D1	U24
D2	U25
D3	U4

REGISTER RENAMING

- Tool: **Register alias table**
  - Entry i.: the ID of the physical register assigned to logical register i
- Procedure:
  - The operands are replaced by physical registers
  - The result is put to a brand new (unused) physical register
  - The register alias table is updated

Original:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
```

After renaming:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
```

Register alias table:

Logical reg.	Physical reg.
R0	T21
R1	T3
R2	T46
R3	T8
D0	U9
D1	U24
D2	U25
D3	**U26**

REGISTER RENAMING

- Tool: **Register alias table**
  - Entry i.: the ID of the physical register assigned to logical register i
- Procedure:
  - The operands are replaced by physical registers
  - The result is put to a brand new (unused) physical register
  - The register alias table is updated

Original:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
i3:  MEM[R2] ← D3
```

After renaming:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
i3:  MEM[T46] ← U26
```

Register alias table:

Logical reg.	Physical reg.
R0	T21
R1	T3
R2	T46
R3	T8
D0	U9
D1	U24
D2	U25
D3	U26

REGISTER RENAMING

- Tool: **Register alias table**
  - Entry i.: the ID of the physical register assigned to logical register i
- Procedure:
  - The operands are replaced by physical registers
  - The result is put to a brand new (unused) physical register
  - The register alias table is updated

Original:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
i3:  MEM[R2] ← D3
i4:  R1 ← R1 + 8
```

After renaming:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
i3:  MEM[T46] ← U26
i4:  T47 ← T3 + 8
```

Register alias table:

Logical reg.	Physical reg.
R0	T21
R1	T3
R2	T46
R3	T8
D0	U9
D1	U24
D2	U25
D3	U26

REGISTER RENAMING

- Tool: **Register alias table**
  - Entry i.: the ID of the physical register assigned to logical register i
- Procedure:
  - The operands are replaced by physical registers
  - The result is put to a brand new (unused) physical register
  - The register alias table is updated

Original:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
i3:  MEM[R2] ← D3
i4:  R1 ← R1 + 8
```

After renaming:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
i3:  MEM[T46] ← U26
i4:  T47 ← T3 + 8
```

Register alias table:

Logical reg.	Physical reg.
R0	T21
R1	**T47**
R2	T46
R3	T8
D0	U9
D1	U24
D2	U25
D3	U26

REGISTER RENAMING

- Tool: **Register alias table**
  - Entry i.: the ID of the physical register assigned to logical register i
- Procedure:
  - The operands are replaced by physical registers
  - The result is put to a brand new (unused) physical register
  - The register alias table is updated

Original:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
i3:  MEM[R2] ← D3
i4:  R1 ← R1 + 8
i5:  R2 ← R2 + 8
```

After renaming:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
i3:  MEM[T46] ← U26
i4:  T47 ← T3 + 8
i5:  T48 ← T46 + 8
```

Register alias table:

Logical reg.	Physical reg.
R0	T21
R1	T47
R2	T46
R3	T8
D0	U9
D1	U24
D2	U25
D3	U26

REGISTER RENAMING

- Tool: **Register alias table**
  - Entry i.: the ID of the physical register assigned to logical register i
- Procedure:
  - The operands are replaced by physical registers
  - The result is put to a brand new (unused) physical register
  - The register alias table is updated

Original:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
i3:  MEM[R2] ← D3
i4:  R1 ← R1 + 8
i5:  R2 ← R2 + 8
```

After renaming:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
i3:  MEM[T46] ← U26
i4:  T47 ← T3 + 8
i5:  T48 ← T46 + 8
```

Register alias table:

Logical reg.	Physical reg.
R0	T21
R1	T47
R2	**T48**
R3	T8
D0	U9
D1	U24
D2	U25
D3	U26

REGISTER RENAMING

- Tool: **Register alias table**
  - Entry i.: the ID of the physical register assigned to logical register i
- Procedure:
  - The operands are replaced by physical registers
  - The result is put to a brand new (unused) physical register
  - The register alias table is updated

Original:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
i3:  MEM[R2] ← D3
i4:  R1 ← R1 + 8
i5:  R2 ← R2 + 8

i6:  D2 ← MEM[R1]
```

After renaming:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
i3:  MEM[T46] ← U26
i4:  T47 ← T3 + 8
i5:  T48 ← T46 + 8

i6:  U27 ← MEM[T47]
```

Register alias table:

Logical reg.	Physical reg.
R0	T21
R1	T47
R2	T48
R3	T8
D0	U9
D1	U24
D2	U25
D3	U26

REGISTER RENAMING

- Tool: **Register alias table**
  - Entry i.: the ID of the physical register assigned to logical register i
- Procedure:
  - The operands are replaced by physical registers
  - The result is put to a brand new (unused) physical register
  - The register alias table is updated

Original:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
i3:  MEM[R2] ← D3
i4:  R1 ← R1 + 8
i5:  R2 ← R2 + 8

i6:  D2 ← MEM[R1]
```

After renaming:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
i3:  MEM[T46] ← U26
i4:  T47 ← T3 + 8
i5:  T48 ← T46 + 8

i6:  U27 ← MEM[T47]
```

Register alias table:

Logical reg.	Physical reg.
R0	T21
R1	T47
R2	T48
R3	T8
D0	U9
D1	U24
D2	**U27**
D3	U26

REGISTER RENAMING

- Tool: **Register alias table**
  - Entry i.: the ID of the physical register assigned to logical register i
- Procedure:
  - The operands are replaced by physical registers
  - The result is put to a brand new (unused) physical register
  - The register alias table is updated

Original:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
i3:  MEM[R2] ← D3
i4:  R1 ← R1 + 8
i5:  R2 ← R2 + 8

i6:  D2 ← MEM[R1]
i7:  D3 ← D2 * D0
```

After renaming:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
i3:  MEM[T46] ← U26
i4:  T47 ← T3 + 8
i5:  T48 ← T46 + 8

i6:  U27 ← MEM[T47]
i7:  U28 ← U27 * U9
```

Register alias table:

Logical reg.	Physical reg.
R0	T21
R1	T47
R2	T48
R3	T8
D0	U9
D1	U24
D2	U27
D3	U26

REGISTER RENAMING

- Tool: **Register alias table**
  - Entry i.: the ID of the physical register assigned to logical register i
- Procedure:
  - The operands are replaced by physical registers
  - The result is put to a brand new (unused) physical register
  - The register alias table is updated

Original:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
i3:  MEM[R2] ← D3
i4:  R1 ← R1 + 8
i5:  R2 ← R2 + 8

i6:  D2 ← MEM[R1]
i7:  D3 ← D2 * D0
```

After renaming:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
i3:  MEM[T46] ← U26
i4:  T47 ← T3 + 8
i5:  T48 ← T46 + 8

i6:  U27 ← MEM[T47]
i7:  U28 ← U27 * U9
```

Register alias table:

Logical reg.	Physical reg.
R0	T21
R1	T47
R2	T48
R3	T8
D0	U9
D1	U24
D2	U27
D3	**U28**

REGISTER RENAMING

- Tool: **Register alias table**
  - Entry i.: the ID of the physical register assigned to logical register i
- Procedure:
  - The operands are replaced by physical registers
  - The result is put to a brand new (unused) physical register
  - The register alias table is updated

Original:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
i3:  MEM[R2] ← D3
i4:  R1 ← R1 + 8
i5:  R2 ← R2 + 8

i6:  D2 ← MEM[R1]
i7:  D3 ← D2 * D0
i8:  MEM[R2] ← D3
```

After renaming:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
i3:  MEM[T46] ← U26
i4:  T47 ← T3 + 8
i5:  T48 ← T46 + 8

i6:  U27 ← MEM[T47]
i7:  U28 ← U27 * U9
i8:  MEM[T48] ← U28
```

Register alias table:

Logical reg.	Physical reg.
R0	T21
R1	T47
R2	T48
R3	T8
D0	U9
D1	U24
D2	U27
D3	U28

REGISTER RENAMING

- Tool: **Register alias table**
  - Entry i.: the ID of the physical register assigned to logical register i
- Procedure:
  - The operands are replaced by physical registers
  - The result is put to a brand new (unused) physical register
  - The register alias table is updated

Original:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
i3:  MEM[R2] ← D3
i4:  R1 ← R1 + 8
i5:  R2 ← R2 + 8

i6:  D2 ← MEM[R1]
i7:  D3 ← D2 * D0
i8:  MEM[R2] ← D3
i9:  R1 ← R1 + 8
```

After renaming:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
i3:  MEM[T46] ← U26
i4:  T47 ← T3 + 8
i5:  T48 ← T46 + 8

i6:  U27 ← MEM[T47]
i7:  U28 ← U27 * U9
i8:  MEM[T48] ← U28
i9:  T49 ← T47 + 8
```

Register alias table:

Logical reg.	Physical reg.
R0	T21
R1	T47
R2	T48
R3	T8
D0	U9
D1	U24
D2	U27
D3	U28

REGISTER RENAMING

- Tool: **Register alias table**
  - Entry i.: the ID of the physical register assigned to logical register i
- Procedure:
  - The operands are replaced by physical registers
  - The result is put to a brand new (unused) physical register
  - The register alias table is updated

Original:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
i3:  MEM[R2] ← D3
i4:  R1 ← R1 + 8
i5:  R2 ← R2 + 8

i6:  D2 ← MEM[R1]
i7:  D3 ← D2 * D0
i8:  MEM[R2] ← D3
i9:  R1 ← R1 + 8
```

After renaming:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
i3:  MEM[T46] ← U26
i4:  T47 ← T3 + 8
i5:  T48 ← T46 + 8

i6:  U27 ← MEM[T47]
i7:  U28 ← U27 * U9
i8:  MEM[T48] ← U28
i9:  T49 ← T47 + 8
```

Register alias table:

Logical reg.	Physical reg.
R0	T21
R1	**T49**
R2	T48
R3	T8
D0	U9
D1	U24
D2	U27
D3	U28

REGISTER RENAMING

- Tool: **Register alias table**
  - Entry i.: the ID of the physical register assigned to logical register i
- Procedure:
  - The operands are replaced by physical registers
  - The result is put to a brand new (unused) physical register
  - The register alias table is updated

Original:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
i3:  MEM[R2] ← D3
i4:  R1 ← R1 + 8
i5:  R2 ← R2 + 8

i6:  D2 ← MEM[R1]
i7:  D3 ← D2 * D0
i8:  MEM[R2] ← D3
i9:  R1 ← R1 + 8
i10: R2 ← R2 + 8
```

After renaming:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
i3:  MEM[T46] ← U26
i4:  T47 ← T3 + 8
i5:  T48 ← T46 + 8

i6:  U27 ← MEM[T47]
i7:  U28 ← U27 * U9
i8:  MEM[T48] ← U28
i9:  T49 ← T47 + 8
i10: T50 ← T48 + 8
```

Register alias table:

Logical reg.	Physical reg.
R0	T21
R1	T49
R2	T48
R3	T8
D0	U9
D1	U24
D2	U27
D3	U28

REGISTER RENAMING

- Tool: **Register alias table**
  - Entry i.: the ID of the physical register assigned to logical register i
- Procedure:
  - The operands are replaced by physical registers
  - The result is put to a brand new (unused) physical register
  - The register alias table is updated

Original:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
i3:  MEM[R2] ← D3
i4:  R1 ← R1 + 8
i5:  R2 ← R2 + 8

i6:  D2 ← MEM[R1]
i7:  D3 ← D2 * D0
i8:  MEM[R2] ← D3
i9:  R1 ← R1 + 8
i10: R2 ← R2 + 8
```

After renaming:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
i3:  MEM[T46] ← U26
i4:  T47 ← T3 + 8
i5:  T48 ← T46 + 8

i6:  U27 ← MEM[T47]
i7:  U28 ← U27 * U9
i8:  MEM[T48] ← U28
i9:  T49 ← T47 + 8
i10: T50 ← T48 + 8
```

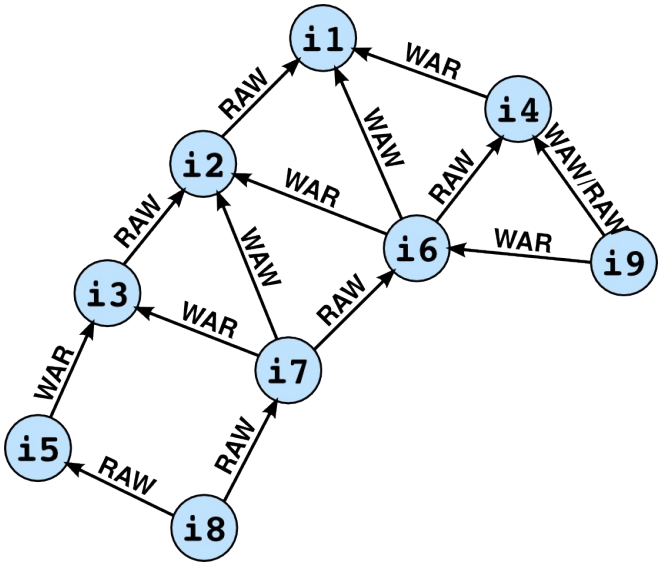
Register alias table:

Logical reg.	Physical reg.
R0	T21
R1	T49
R2	**T50**
R3	T8
D0	U9
D1	U24
D2	U27
D3	U28

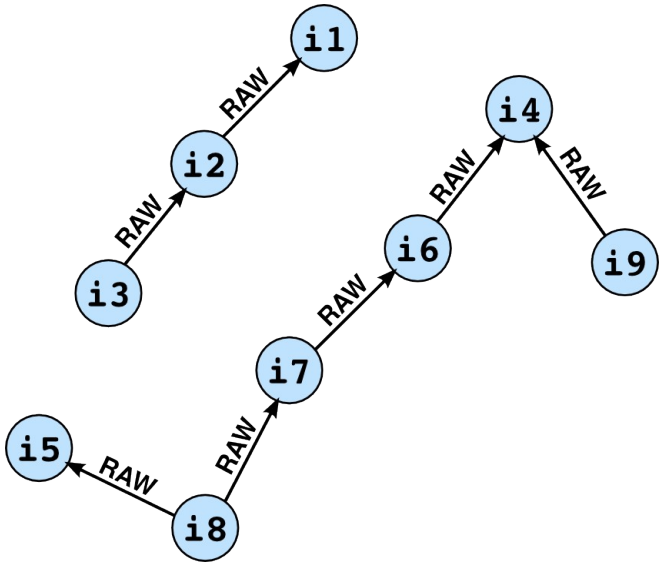
REGISTER RENAMING

- Result:
  - All WAW and WAR dependencies disappeared
  - ...since we always store the result in a “clean”, unused register
- The precedence graph:

Before register renaming:



After register renaming:

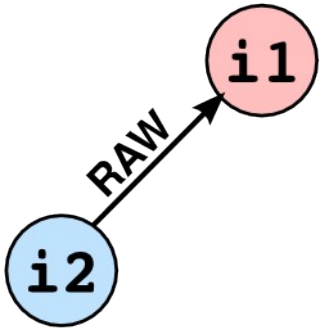


EXAMPLE

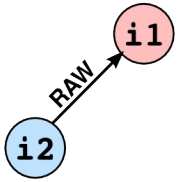

```
i1:   U25 ← MEM[T3]
i2:   U26 ← U25 * U9
i3:   MEM[T46] ← U26
i4:   T47 ← T3 + 8
i5:   T48 ← T46 + 8

i6:   U27 ← MEM[T47]
i7:   U28 ← U27 * U9
i8:   MEM[T48] ← U28
i9:   T49 ← T47 + 8
i10:  T50 ← T48 + 8
```


```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

EXAMPLE

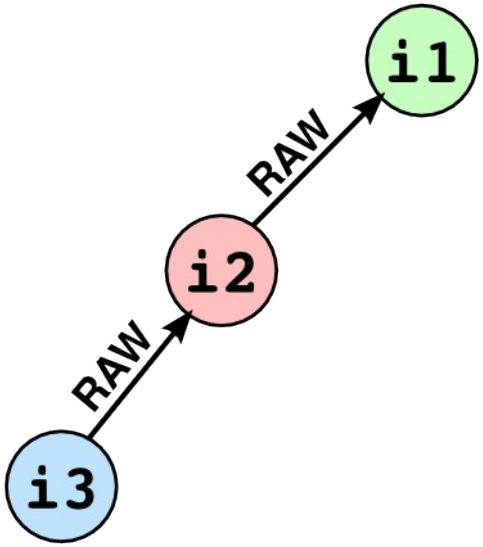



```
i1:   U25 ← MEM[T3]
i2:   U26 ← U25 * U9
i3:   MEM[T46] ← U26
i4:   T47 ← T3 + 8
i5:   T48 ← T46 + 8

i6:   U27 ← MEM[T47]
i7:   U28 ← U27 * U9
i8:   MEM[T48] ← U28
i9:   T49 ← T47 + 8
i10:  T50 ← T48 + 8
```

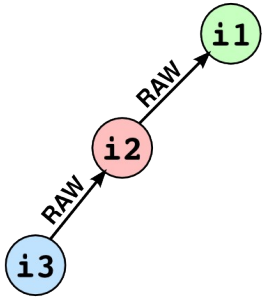



```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

EXAMPLE

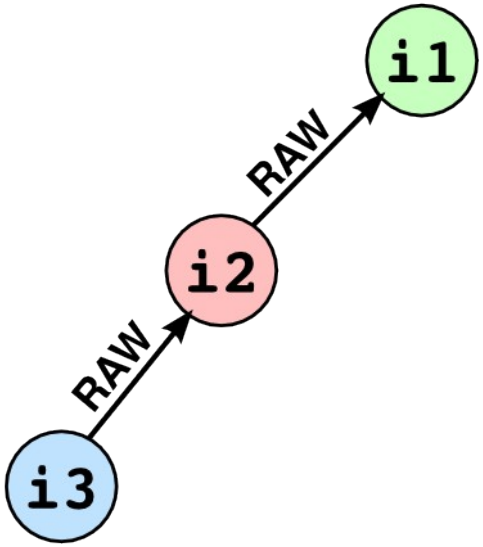



```
i1:   U25 ← MEM[T3]
i2:   U26 ← U25 * U9
i3:   MEM[T46] ← U26
i4:   T47 ← T3 + 8
i5:   T48 ← T46 + 8

i6:   U27 ← MEM[T47]
i7:   U28 ← U27 * U9
i8:   MEM[T48] ← U28
i9:   T49 ← T47 + 8
i10:  T50 ← T48 + 8
```

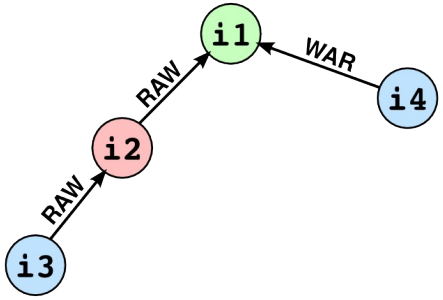



```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

EXAMPLE

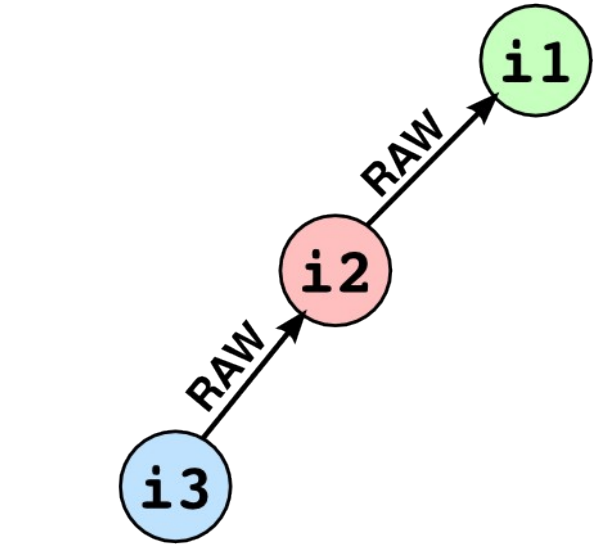


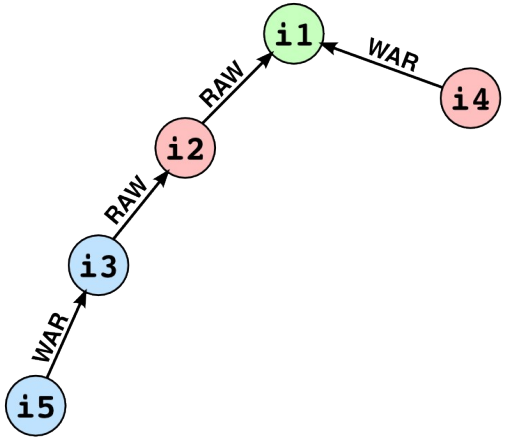

```
i1:   U25 ← MEM[T3]
i2:   U26 ← U25 * U9
i3:   MEM[T46] ← U26
i4:   T47 ← T3 + 8
i5:   T48 ← T46 + 8

i6:   U27 ← MEM[T47]
i7:   U28 ← U27 * U9
i8:   MEM[T48] ← U28
i9:   T49 ← T47 + 8
i10:  T50 ← T48 + 8
```




```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

EXAMPLE

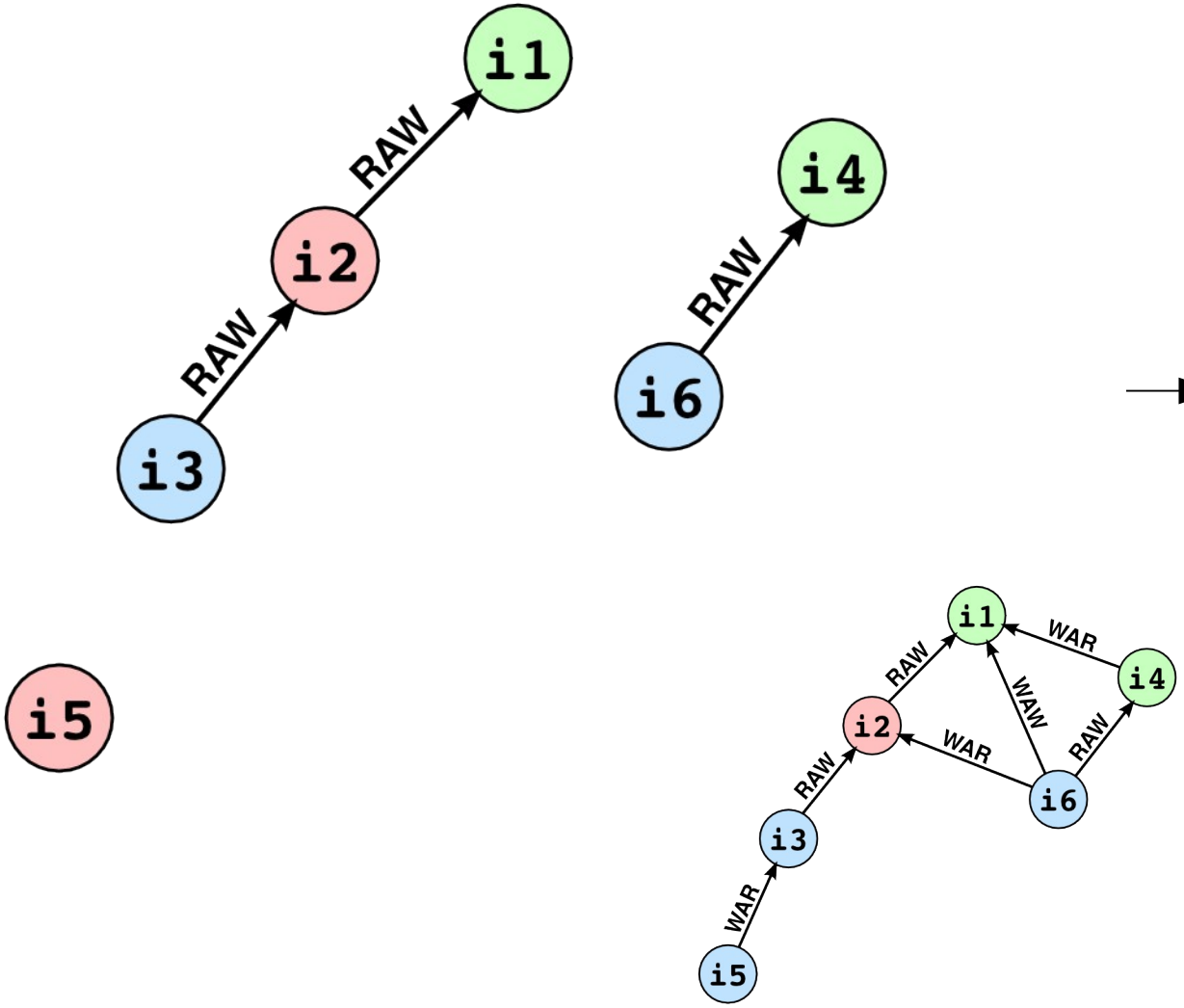





```
i1:   U25 ← MEM[T3]
i2:   U26 ← U25 * U9
i3:   MEM[T46] ← U26
i4:   T47 ← T3 + 8
i5:   T48 ← T46 + 8

i6:   U27 ← MEM[T47]
i7:   U28 ← U27 * U9
i8:   MEM[T48] ← U28
i9:   T49 ← T47 + 8
i10:  T50 ← T48 + 8
```

```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

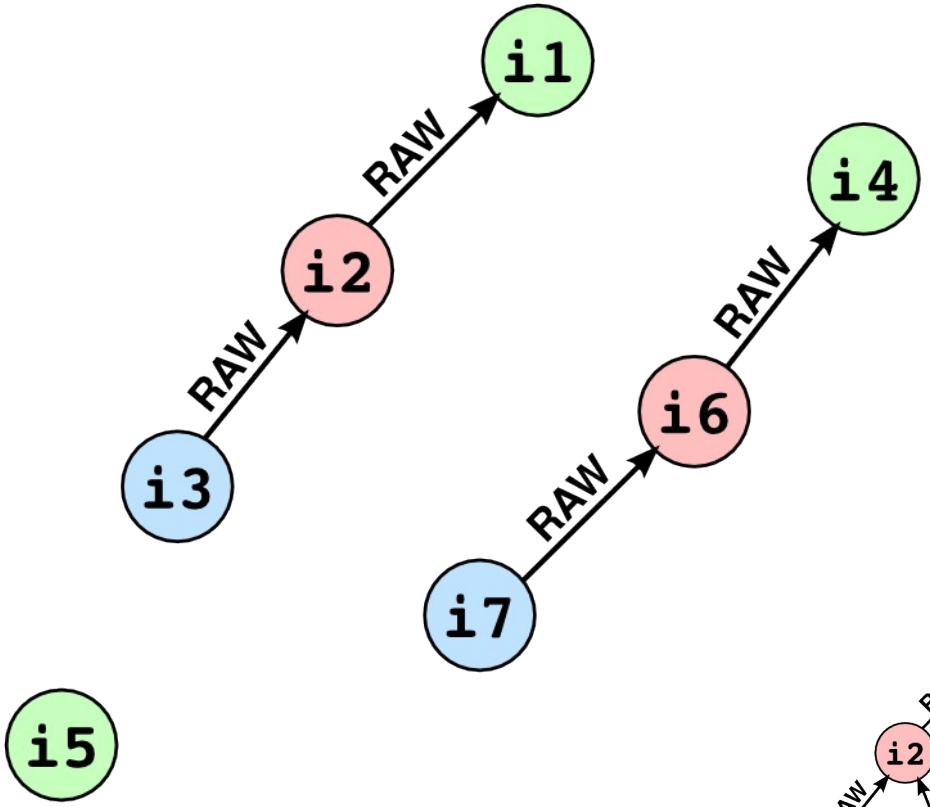
EXAMPLE



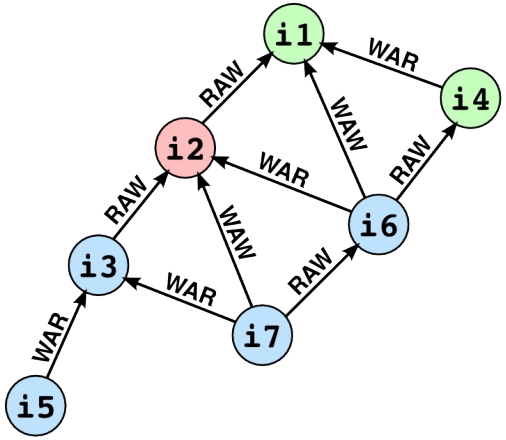

```
i1:   U25 ← MEM[T3]
i2:   U26 ← U25 * U9
i3:   MEM[T46] ← U26
i4:   T47 ← T3 + 8
i5:   T48 ← T46 + 8

i6:   U27 ← MEM[T47]
i7:   U28 ← U27 * U9
i8:   MEM[T48] ← U28
i9:   T49 ← T47 + 8
i10:  T50 ← T48 + 8
```

```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

EXAMPLE

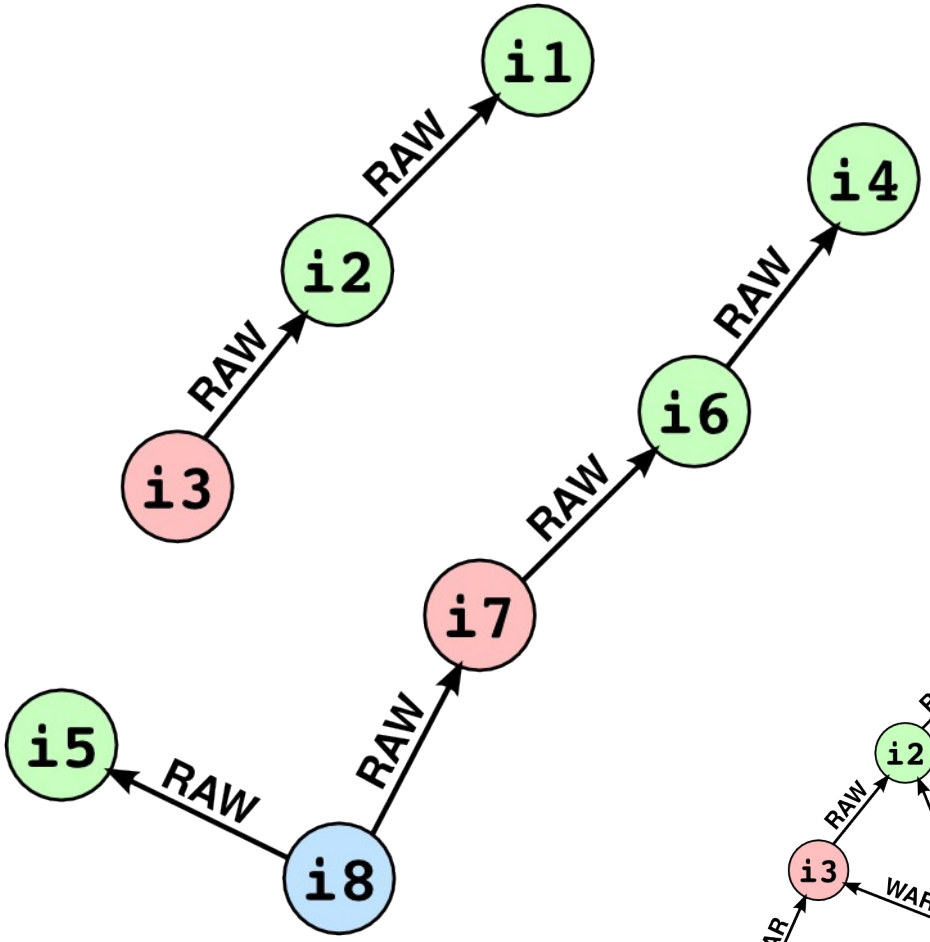



```
i1:   U25 ← MEM[T3]
i2:   U26 ← U25 * U9
i3:   MEM[T46] ← U26
i4:   T47 ← T3 + 8
i5:   T48 ← T46 + 8

i6:   U27 ← MEM[T47]
i7:   U28 ← U27 * U9
i8:   MEM[T48] ← U28
i9:   T49 ← T47 + 8
i10:  T50 ← T48 + 8
```

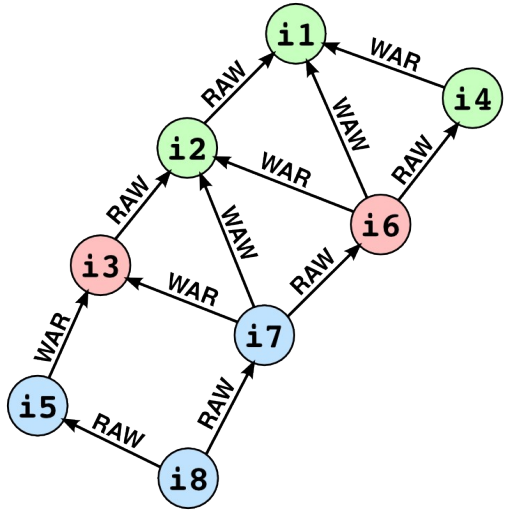



```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

EXAMPLE




```
i1:   U25 ← MEM[T3]
i2:   U26 ← U25 * U9
i3:   MEM[T46] ← U26
i4:   T47 ← T3 + 8
i5:   T48 ← T46 + 8

i6:   U27 ← MEM[T47]
i7:   U28 ← U27 * U9
i8:   MEM[T48] ← U28
i9:   T49 ← T47 + 8
i10:  T50 ← T48 + 8
```




```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

REGISTER RENAMING

- Conclusion:
  - Effective!
  - The larger the instruction window is, the more efficient the register renaming is.

THE INGREDIENTS OF OUT-OF-ORDER EXECUTION

- The CPU has to fetch several instructions to be able to re-order them
  - They must be stored somewhere:

    → **instruction window (instruction pool, reservation station)**
- A procedure that determines the execution order of the instructions

  → **dynamic scheduler**
- An idea to reduce the number of dependencies between the instructions

  → **register renaming**
- A trick to show in-order execution to the outside world

  → **re-order buffer**

THE INGREDIENTS OF OUT-OF-ORDER EXECUTION

- The CPU has to fetch several instructions to be able to re-order them
  - They must be stored somewhere:

    → **instruction window (instruction pool, reservation station)**
- A procedure that determines the execution order of the instructions

  → **dynamic scheduler**
- An idea to reduce the number of dependencies between the instructions

  → **register renaming**
- A trick to show in-order execution to the outside world

  → **re-order buffer**

RE-ORDER BUFFER

- Unwanted side effect of out-of-order execution:
  - Registers/memory does not change in the order as given by the program
  - The memory is shared
    - Other processors are using it,
    - I/O devices are using it,
    - etc.
  - Not everybody is prepared for this behavior!
- Solution: re-order buffer (ROB)
  - An entry is allocated to each instruction when entering the CPU
  - After executing the instruction, the ROB stores
    - The result of the instruction
    - The destination of the result (register/memory location)
  - The results are written to the destination only when all preceding instructions are ready
    - Complete: execution has been finished
    - Retire: the result is written to its destination

EXAMPLE

Instruction:	Ready?	Result:	Where?
U25 ← MEM[T3]			U25

EXAMPLE

Instruction:	Ready?	Result:	Where?
U25 ← MEM[T3]			U25
U26 ← U25 * U9			U26

EXAMPLE

Instruction:	Ready?	Result:	Where?
U25 ← MEM[T3]	✔	<value>	U25
U26 ← U25 * U9			U26
MEM[T46] ← U26			MEM[T46]

EXAMPLE

Instruction:	Ready?	Result:	Where?
U25 ← MEM[T3]	✔	<value>	U25
U26 ← U25 * U9			U26
MEM[T46] ← U26			MEM[T46]
T47 ← T3 + 8			T47

EXAMPLE

Instruction:	Ready?	Result:	Where?
U25 ← MEM[T3]	✔	<value>	U25
U26 ← U25 * U9			U26
MEM[T46] ← U26			MEM[T46]
T47 ← T3 + 8			T47
T48 ← T46 + 8			T48

EXAMPLE

Instruction:	Ready?	Result:	Where?
U25 ← MEM[T3]	✔	<value>	U25
U26 ← U25 * U9			U26
MEM[T46] ← U26			MEM[T46]
T47 ← T3 + 8	✔	<value>	T47
T48 ← T46 + 8			T48
U27 ← MEM[T47]			U27

EXAMPLE

Instruction:	Ready?	Result:	Where?
U25 ← MEM[T3]	✔	<value>	U25
U26 ← U25 * U9			U26
MEM[T46] ← U26			MEM[T46]
T47 ← T3 + 8	✔	<value>	T47
T48 ← T46 + 8	✔	<value>	T48
U27 ← MEM[T47]			U27
U28 ← U27 * U9			U28

EXAMPLE

Instruction:	Ready?	Result:	Where?
U25 ← MEM[T3]	✔	<value>	U25
U26 ← U25 * U9	✔	<value>	U26
MEM[T46] ← U26			MEM[T46]
T47 ← T3 + 8	✔	<value>	T47
T48 ← T46 + 8	✔	<value>	T48
U27 ← MEM[T47]	✔	<value>	U27
U28 ← U27 * U9			U28
MEM[T48] ← U28			MEM[T48]

EXAMPLE

Instruction:	Ready?	Result:	Where?
U25 ← MEM[T3]	✔	<value>	U25
U26 ← U25 * U9	✔	<value>	U26
MEM[T46] ← U26	✔	<value>	MEM[T46]
T47 ← T3 + 8	✔	<value>	T47
T48 ← T46 + 8	✔	<value>	T48
U27 ← MEM[T47]	✔	<value>	U27
U28 ← U27 * U9			U28
MEM[T48] ← U28			MEM[T48]
T49 ← T47 + 8			T49

OUT-OF-ORDER INSTRUCTION EXECUTION

- Covered topics:
  - Principle: dependency analysis
  - Can be more efficient if register renaming is used
  - We can imitate in-order execution with a re-order buffer
- Implementations:
  - Scoreboard (1964): no register renaming, no ROB
  - Tomasulo (IBM, 1967): has register renaming, but no ROB
  - Since Intel P6 (including Core i7): both register renaming and ROB
  - In mobile devices: since ARM Cortex A9

DEPARTMENT OF
NETWORKED SYSTEMS
AND SERVICES

MŰEGYETEM 1782