Computer Architecture

Dynamic Scheduling to Explore ILP

Dr. Mohammad Reza Selim

Lecture Outline

- Introduction
- What is Dynamic Scheduling
- How Dynamic Scheduling Works
- Register Renaming

Introduction

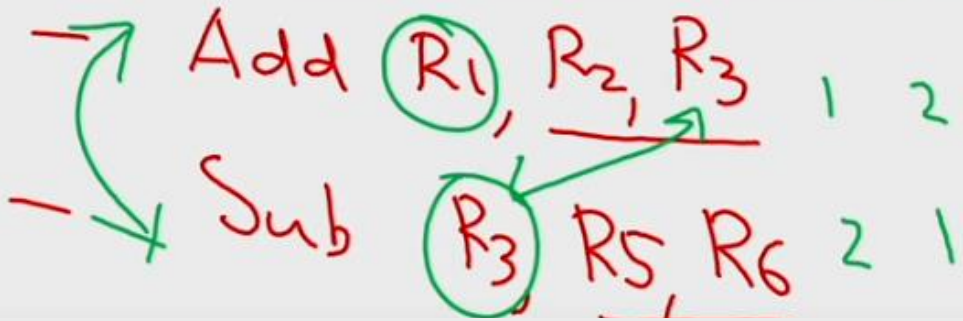
- Pipelining overlaps execution of instructions
- Exploits Instruction Level Parallelism (ILP)
- There are two main approaches:
  - Compiler-based static approaches
  - Hardware-based dynamic approaches
- **Exploiting ILP, goal is to minimize CPI**
  - Pipeline CPI = Ideal (base) CPI + Structural stalls +Data hazard stalls +Control stalls

Data Dependence

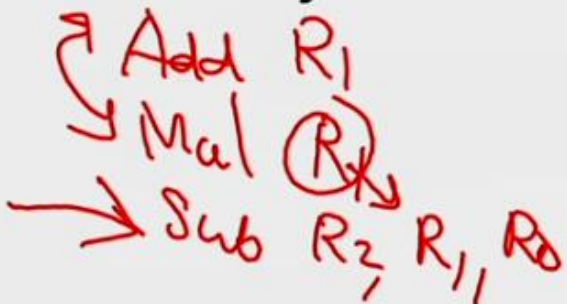
- Loop-Level Parallelism
  - Unroll loop statically or dynamically
- Challenges→ Data dependency
- Data dependence conveys possibility of a hazard
- Dependent instructions cannot be executed simultaneously
- Pipeline determines if dependence is detected and if it causes a stall or not
- Data dependence conveys upper bound on exploitable instruction level parallelism

Name and Output Dependence

- Two instructions use the same name but no flow of information.
- Not a true data dependence, but is a problem when reordering instructions



- **Antidependence:** instruction j writes a register or memory location that instruction i reads
  - Initial ordering (i before j) must be preserved
- **Output dependence:** instruction i and instruction j write the same register or memory location
  - Ordering must be preserved



- To resolve, use renaming techniques

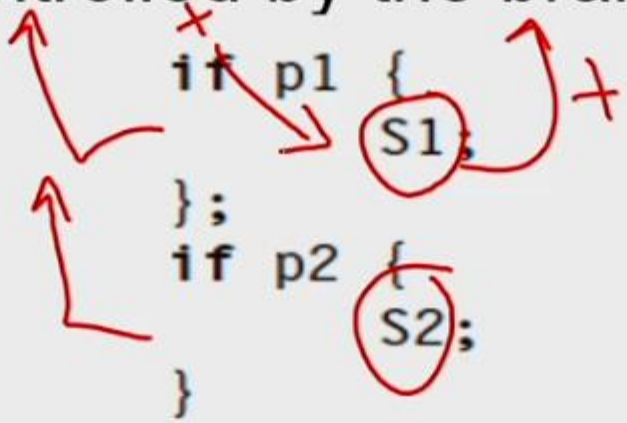
Control Dependence

- Ordering of instruction with respect to a branch instruction
  - Instruction that is control dependent on a branch cannot be moved **before** the branch so that its execution is no longer controller by the branch
  - An instruction that is not control dependent on a branch cannot be moved **after** the branch so that its execution is controlled by the branch.

Example :

```
    DADDU R1,R2,R3
    BEQZ R2,L1
    LW R1, 0(R2)
L1: ...
```

```
if p1 {
    S1;
};
if p2 {
    S2;
}
```



Compiler Techniques to Explore ILP

- Find and overlap sequence of unrelated instruction
- Pipeline scheduling
  - Separate dependent instruction from the source instruction by pipeline latency of the source instruction

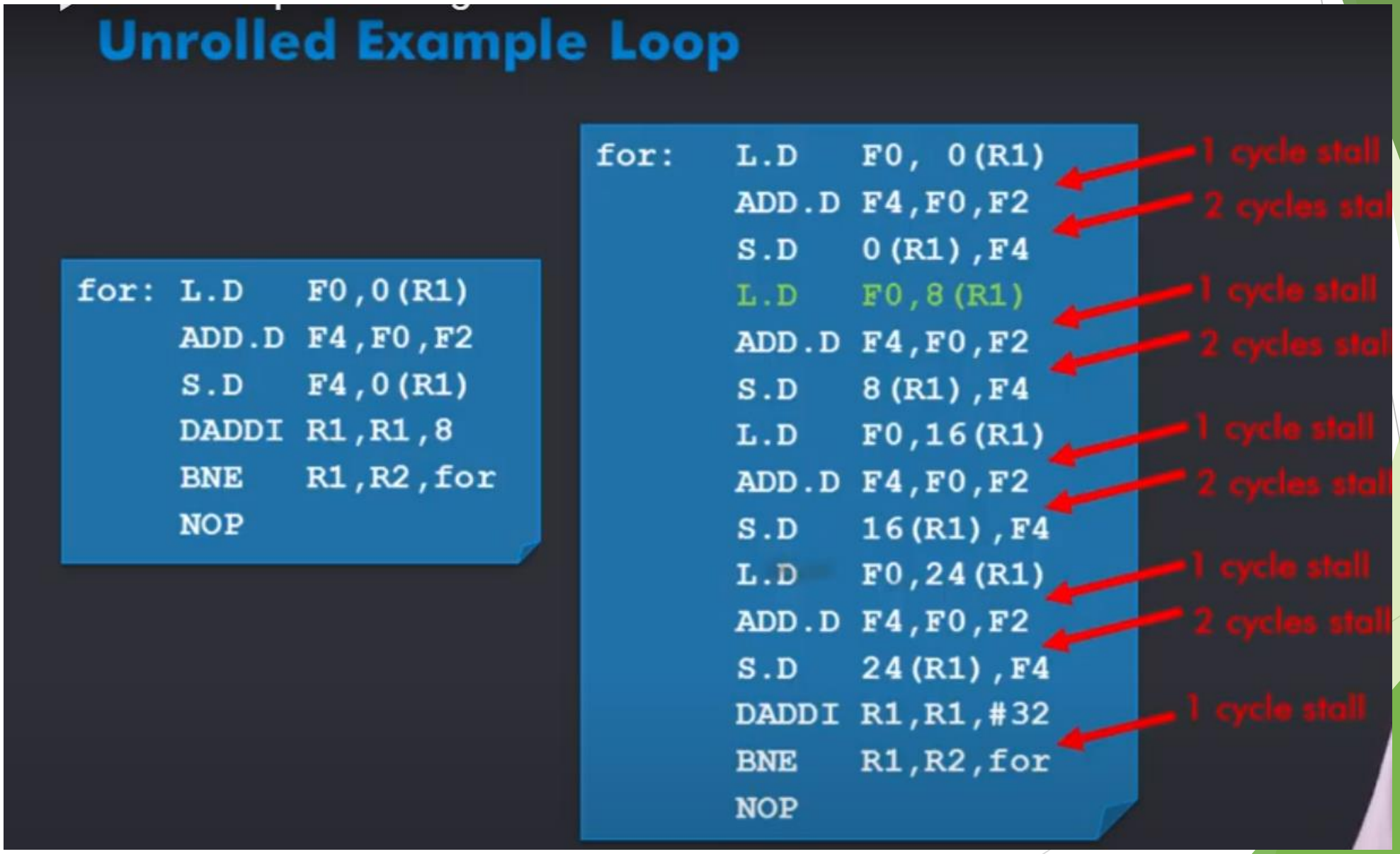
Loop Unrolling (1)

- Replicate loop body multiple times, adjusting loop termination code

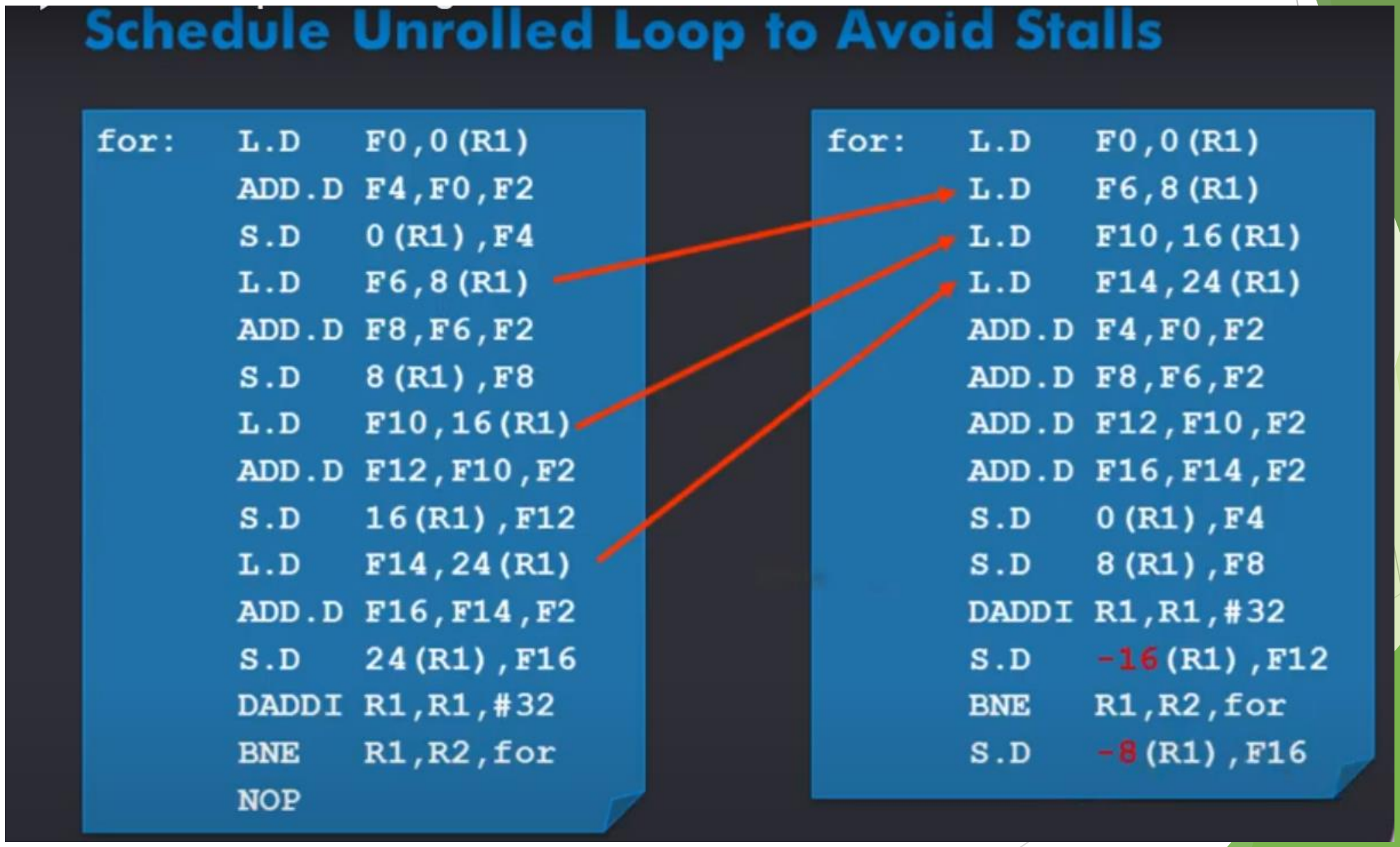
```
for (i=0; i<1000; i++)
        x[i] = x[i]+s;
```

```
for (i=0; i<1000; i+=4){
        x[i] = x[i]+s;
        x[i+1] = x[i+1]+s;
        x[i+2] = x[i+2]+s;
        x[i+3] = x[i+3]+s;
}
```

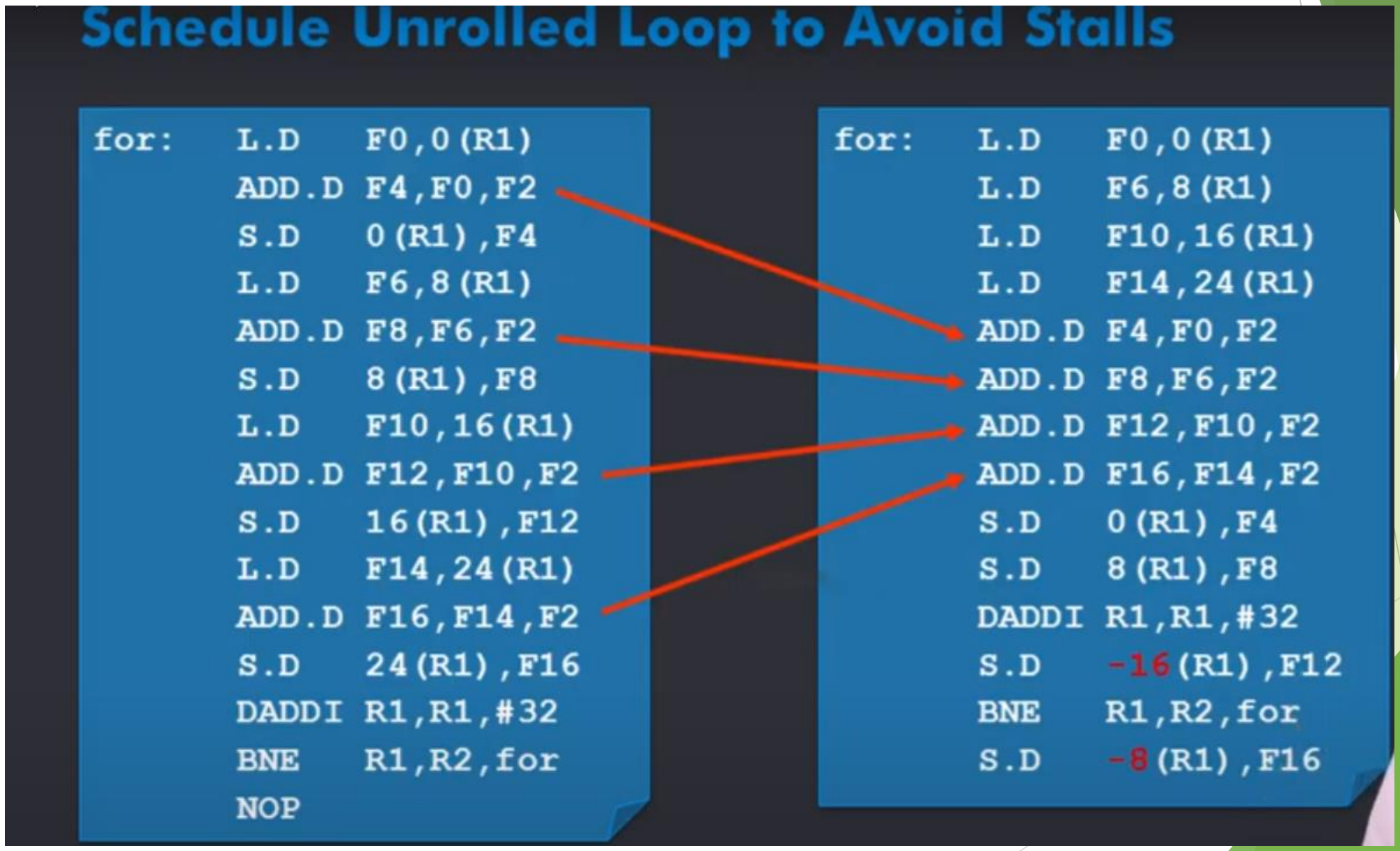
Loop Unrolling (3)



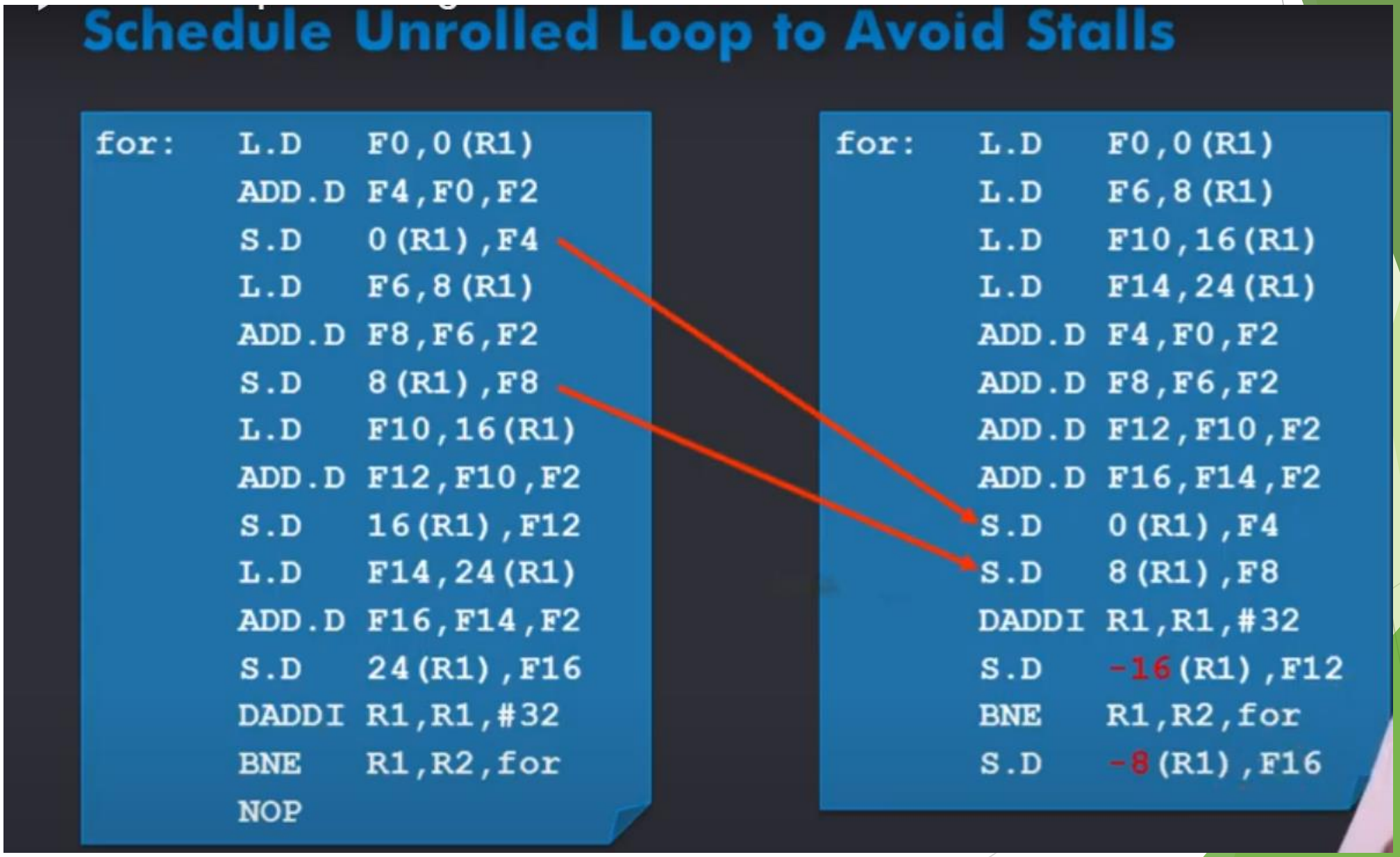
Loop Unrolling (5)



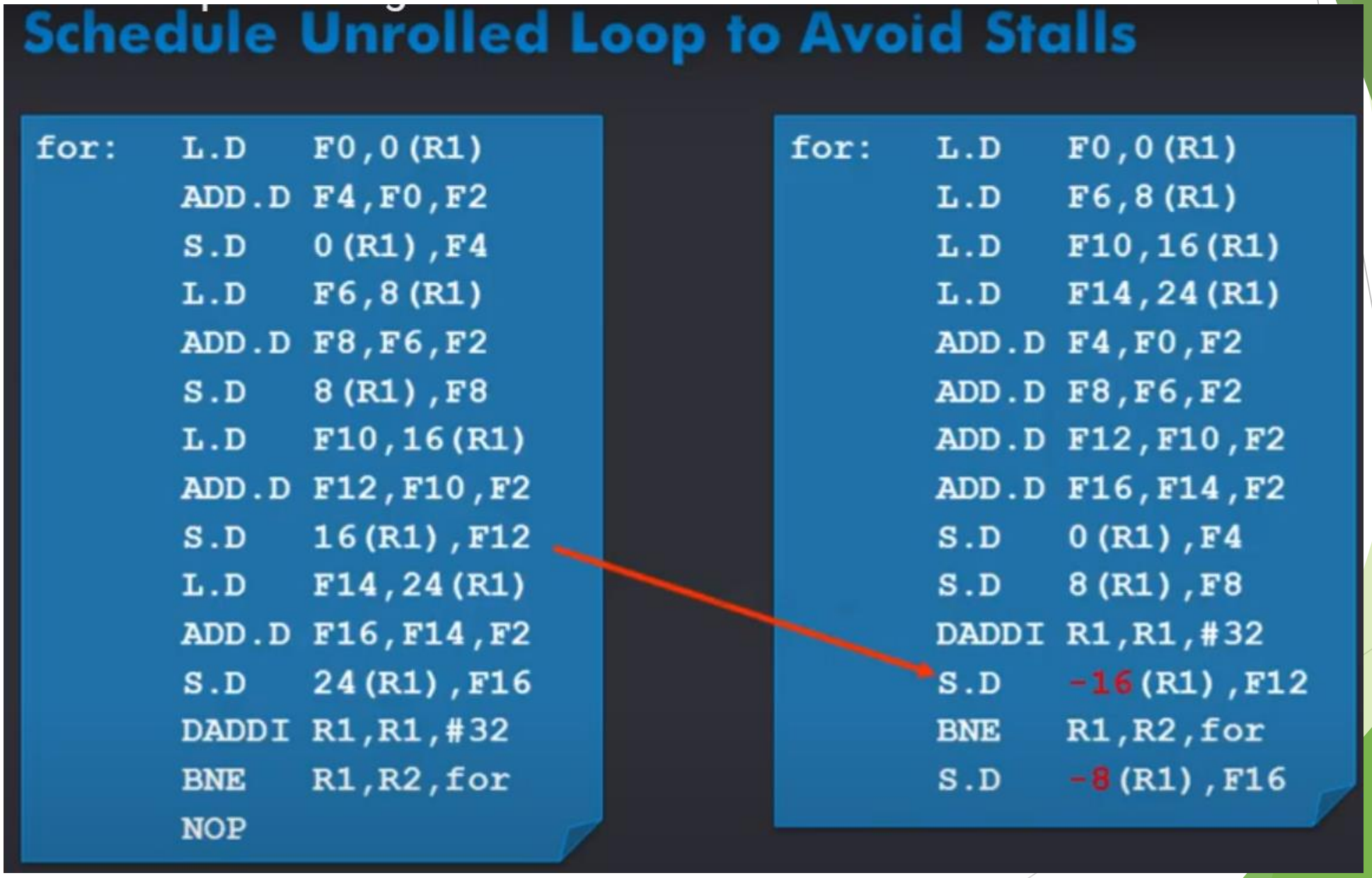
Loop Unrolling (6)



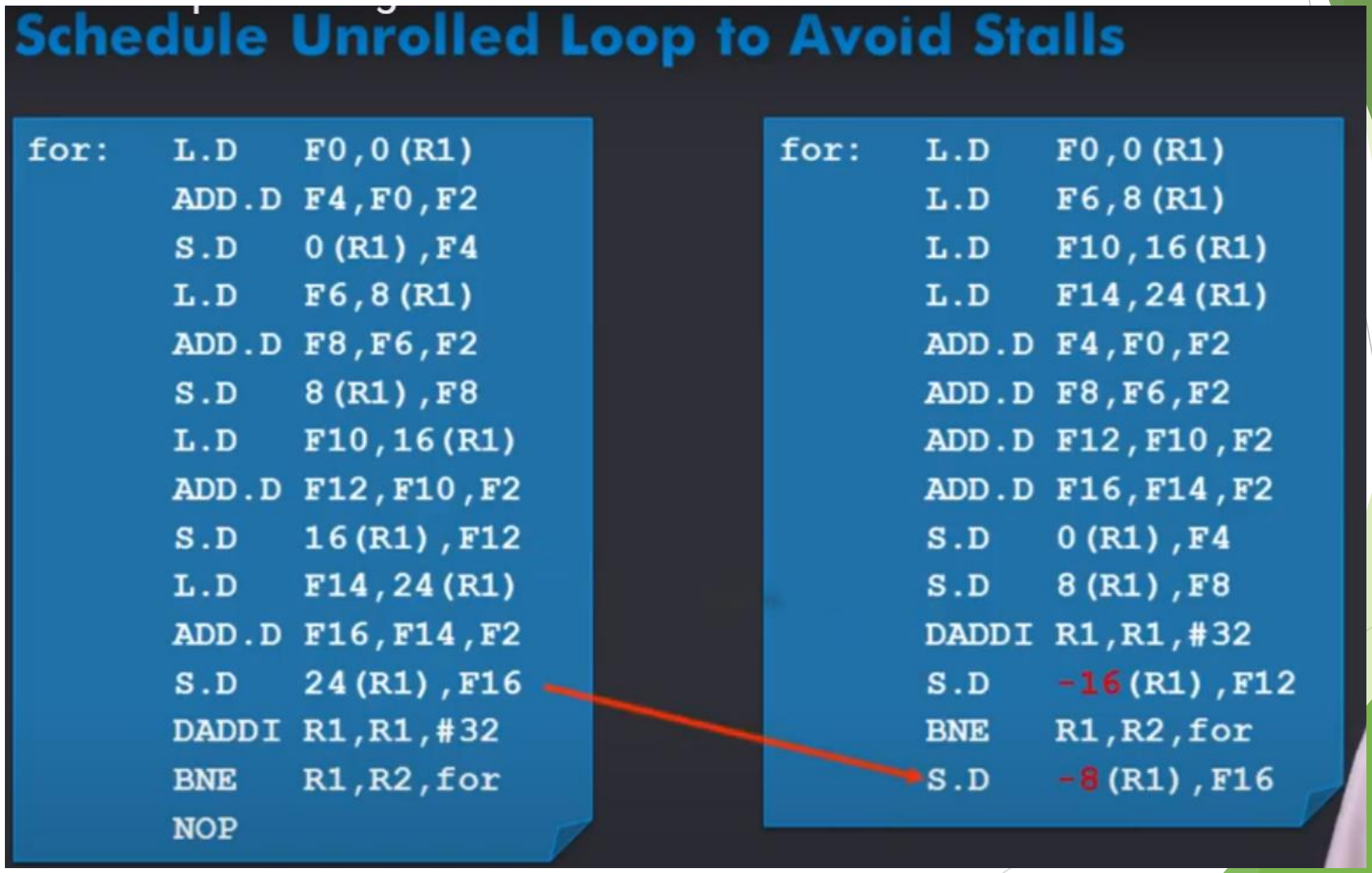
Loop Unrolling (7)



Loop Unrolling (8)



Loop Unrolling (9)



Loop Unrolling (10)

Final Scheduled Code for Example Loop

```
for:   L.D    F0,0(R1)
       L.D    F6,8(R1)
       L.D    F10,16(R1)
       L.D    F14,24(R1)
       ADD.D  F4,F0,F2
       ADD.D  F8,F6,F2
       ADD.D  F12,F10,F2
       ADD.D  F16,F14,F2
       S.D    0(R1),F4
       S.D    8(R1),F8
       DADDI  R1,R1,#32
       S.D    -16(R1),F12
       BNE    R1,R2,for
       S.D    -8(R1),F16
```

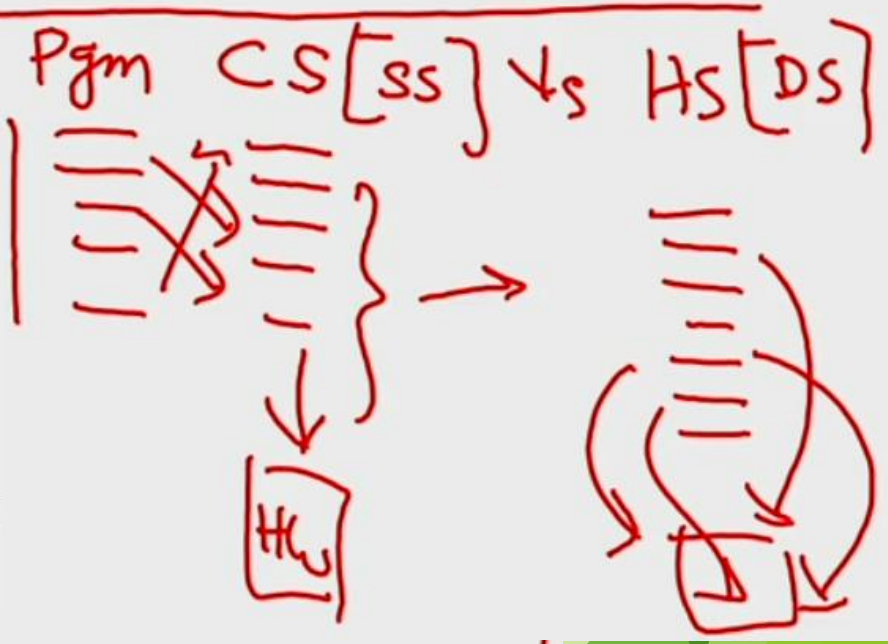
- Runs in 14 cc (no stalls) per iteration
  - 14/4 = 3.5 cc per element
- Made possible by:
  - moving all loads before all ADD.D
  - moving 1 S.D between DADDI and BNE
  - moving 1 S.D in branch delay slot
  - using different registers
- When safe for compiler to do such changes?

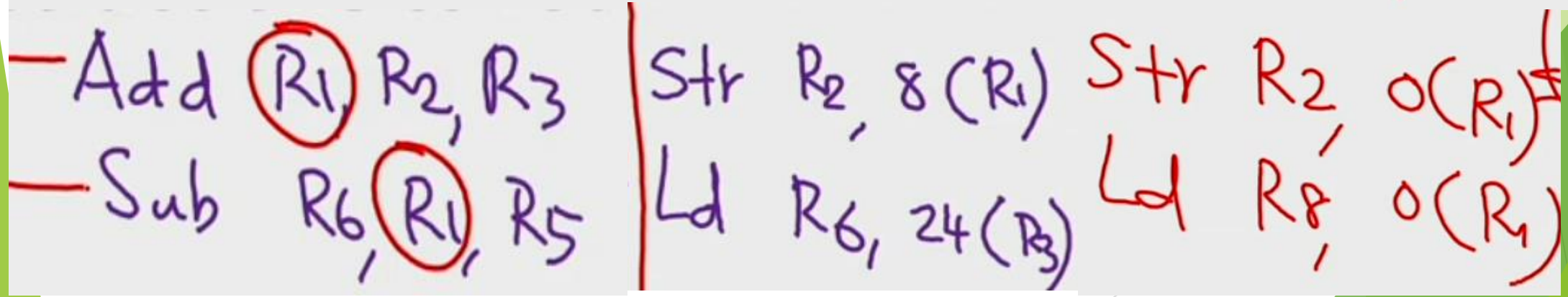
Dynamic Scheduling

- **Rearrange execution order of instructions to reduce stalls while maintaining data flow.**

8 + [R2] == 24 + [R3]

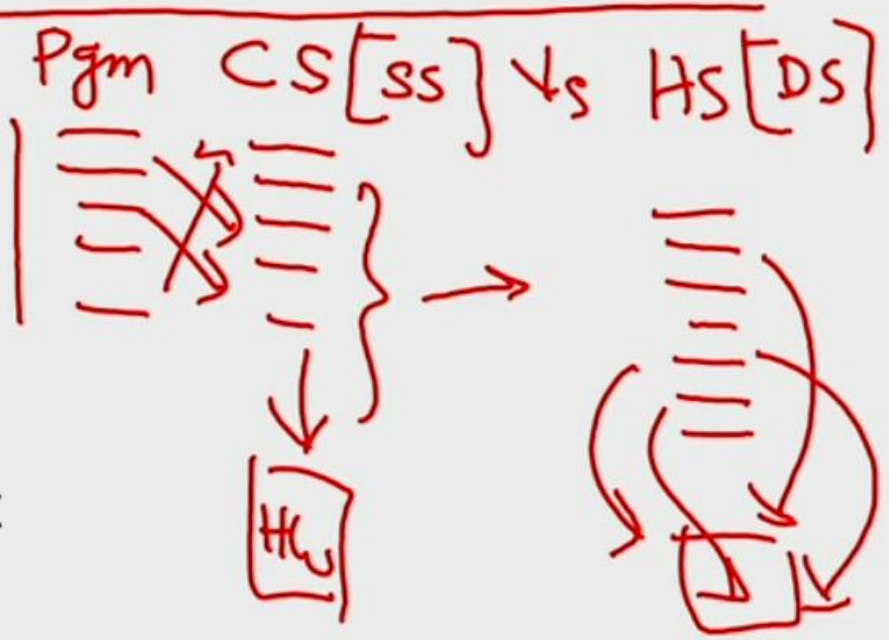
- Advantages:
  - Compiler doesn't need to have knowledge of micro-architecture
  - Handles cases where dependencies are unknown at compile time





Dynamic Scheduling

- **Rearrange execution order of instructions to reduce stalls while maintaining data flow**



- Advantages:
  - Compiler doesn't need to have knowledge of micro-architecture
  - Handles cases where dependencies are unknown at compile time
- Disadvantage:
  - Substantial increase in hardware complexity
  - Complicates exceptions

Static vs Dynamic Scheduling

- With static scheduling the compiler tries to reorder these instructions during compile time to reduce pipeline stalls.
  - Uses less hardware
  - No knowledge of target hardware is needed.
- With dynamic scheduling the hardware tries to rearrange the instructions during run-time to reduce pipeline stalls.
  - Simpler compiler
  - Handles dependencies not known at compile time
  - Allows code compiled for a different machine to run efficiently.

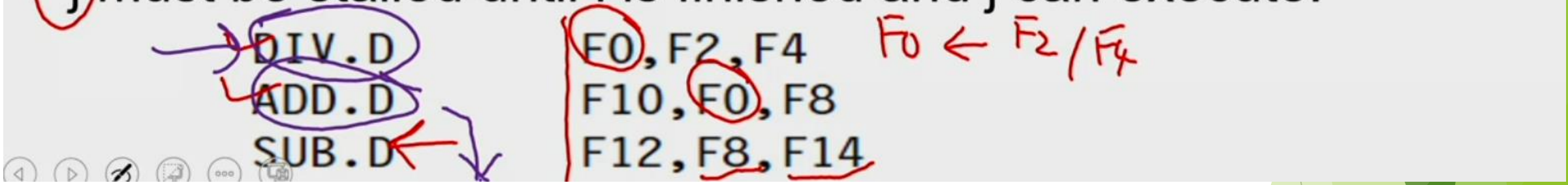
How Dynamic Scheduling Works

- Limitation of simple pipelining. Inorder.
  - In-order instruction issue and execution.
  - Instructions are issued in program order.
  - If an instruction is stalled in the pipeline, no later instructions can proceed.

```
1  add r1,r2,r3
2  sub r4,r1,r3
3  and r6,r1,r7
4  or r8,r1,r9
```

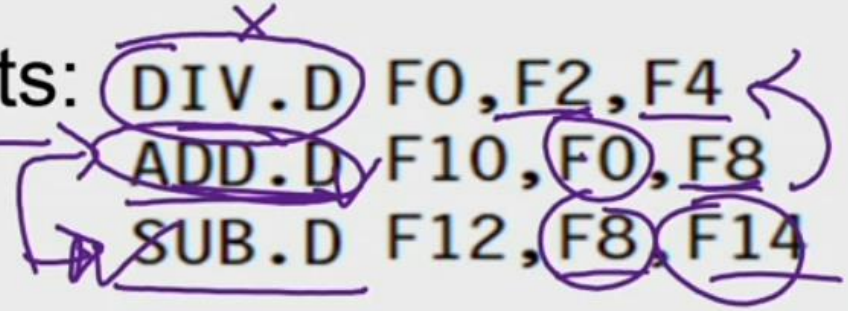
How Dynamic Scheduling Works

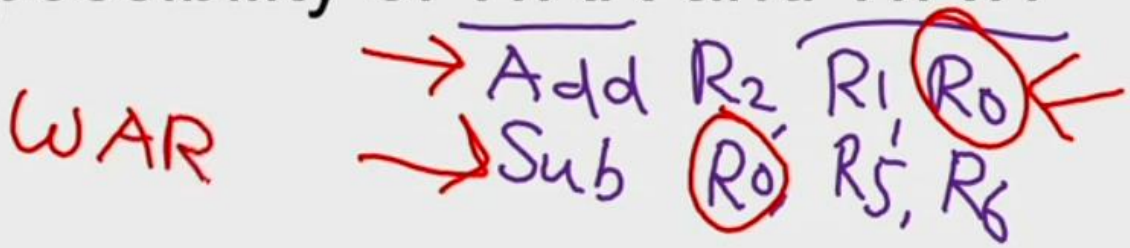
- If instruction j depends on a long-running instruction i currently in execution in the pipeline, then all instructions after j must be stalled until i is finished and j can execute.



How Dynamic Scheduling Works

- Separate the issue process into two parts:
  - checking for any structural hazards.
  - waiting for the absence of a data hazard.
- Use in-order instruction issue but we want an instruction to begin execution as soon as its data operands are available.
- out-of-order execution → out-of-order completion.
- OOO execution introduces the possibility of WAR and WAW hazards

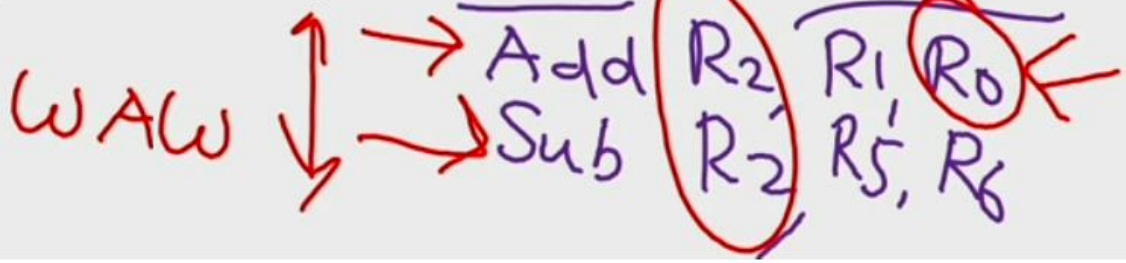




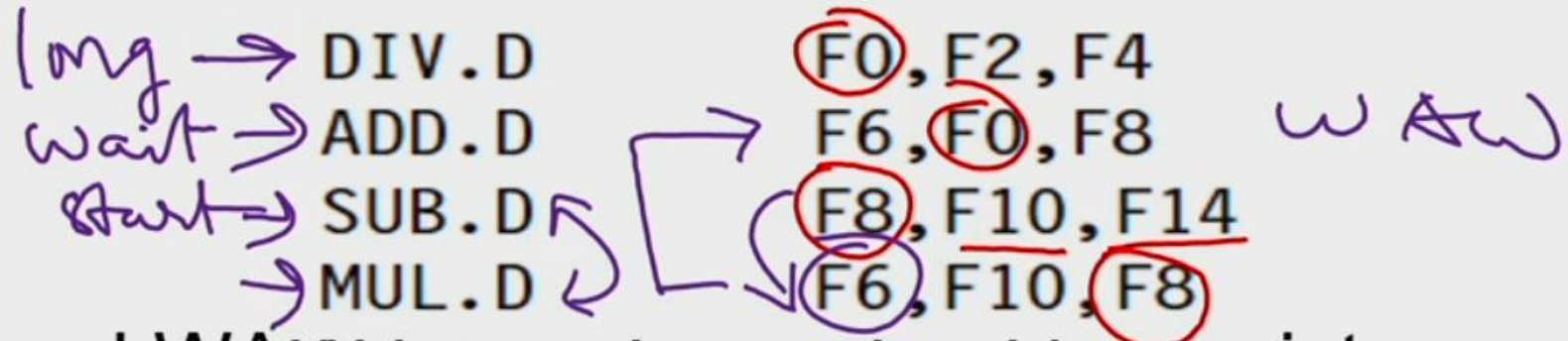
How Dynamic Scheduling Works

- Separate the issue process into two parts:
  - checking for any structural hazards.
  - waiting for the absence of a data hazard.
- Use in-order instruction issue but we want an instruction to begin execution as soon as its data operands are available.
- out-of-order execution → out-of-order completion.
- OOO execution introduces the possibility of WAR and WAW hazards

```
DIV.D  F0,F2,F4
ADD.D  F10,F0,F8
SUB.D  F12,F8,F14
```

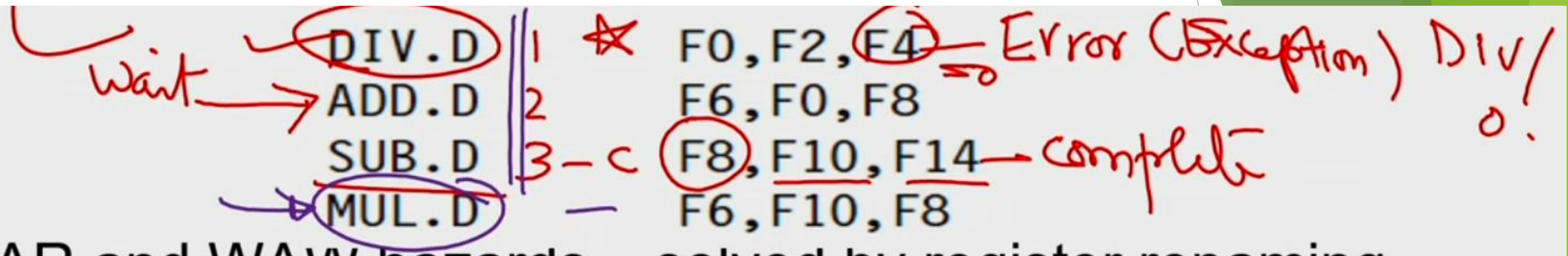


How Dynamic Scheduling Works



- WAR and WAW hazards – solved by register renaming
- Possibility of imprecise exception (2 possibilities).
  - The pipeline may have already completed instructions that are later in program order than the instruction causing the exception.

How Dynamic Scheduling Works

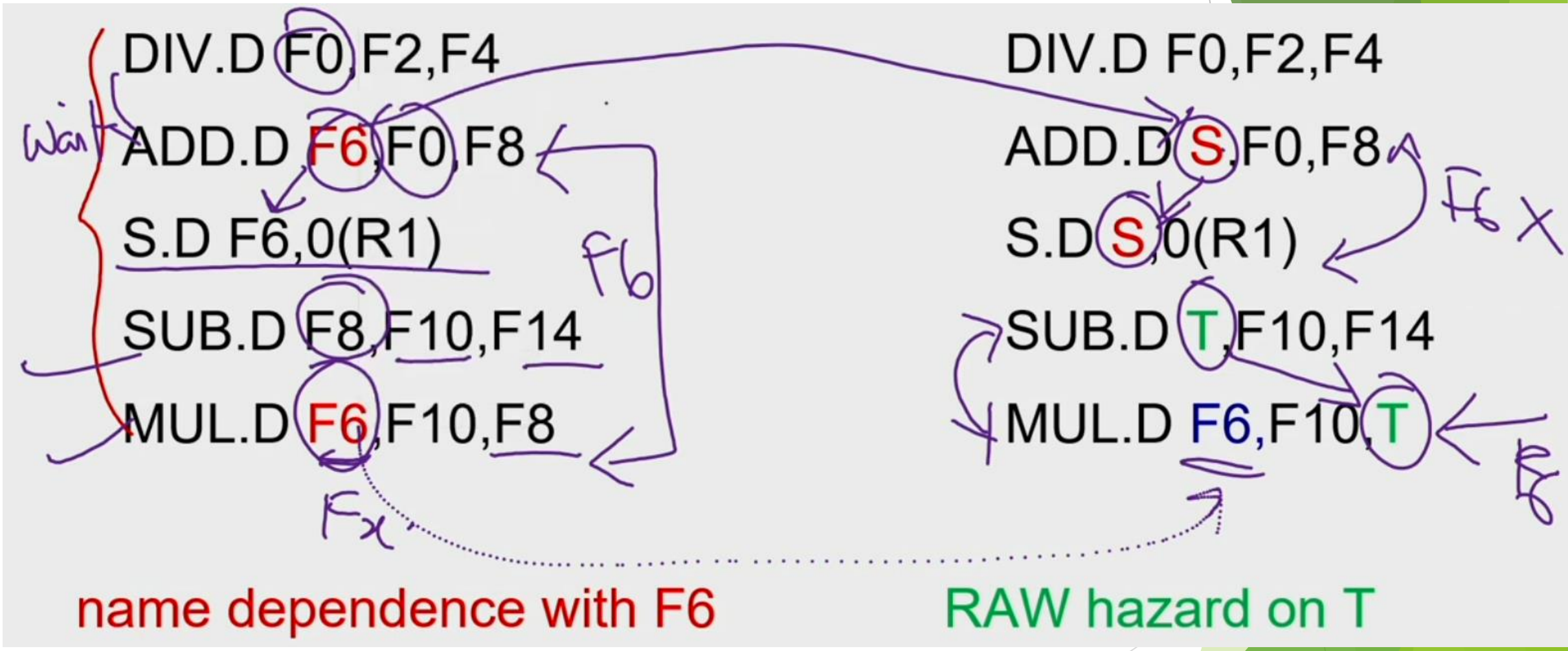


- WAR and WAW hazards – solved by register renaming
- Possibility of imprecise exception (2 possibilities).
  - The pipeline may have already completed instructions that are later in program order than the instruction causing the exception.
  - The pipeline may have not yet completed some instructions that are earlier in program order than the instruction causing the exception

How Dynamic Scheduling Works

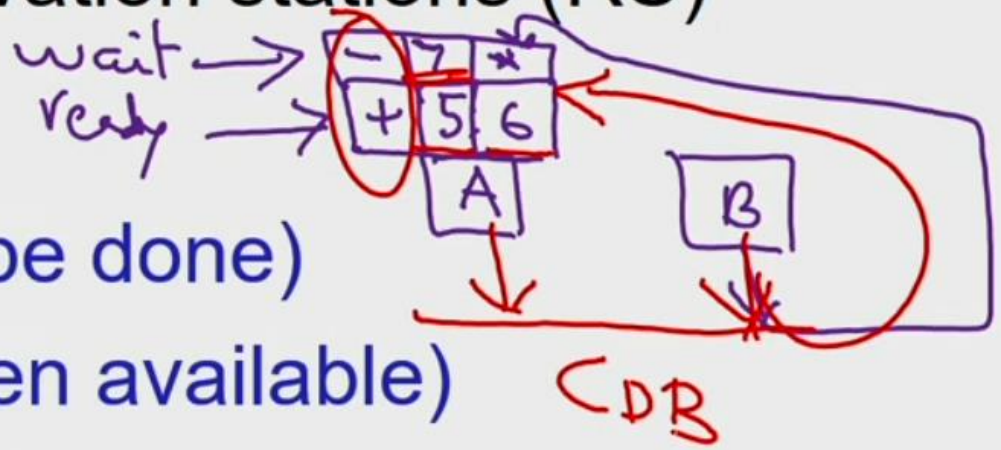
- To allow out-of-order execution, **split the ID stage into two**
  - **Issue**—Decode instructions, check for structural hazards.
  - **Read operands**—Wait until no data hazards, then read operands.
- In a dynamically scheduled pipeline, all instructions pass through the **issue stage in order** (in-order issue); however, they can be stalled or bypass each other in the second stage (read operands) and thus enter execution out of order.
- Done by - **score boarding technique**
- Approach used - **Tomasulo's algorithm**

Register Renaming



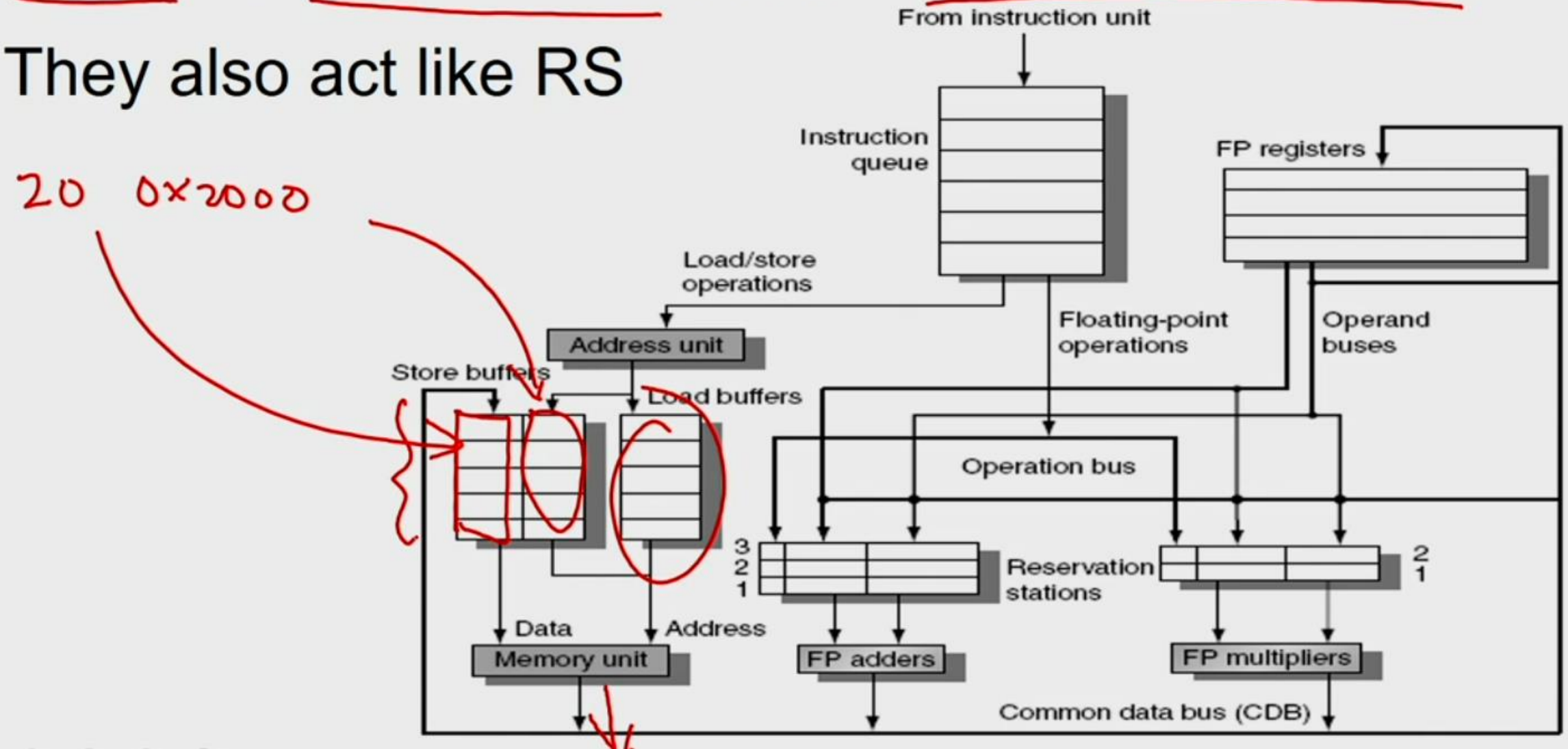
Register Renaming

- Register renaming is done by reservation stations (RS)
- Each RS Contains:
  - The instruction (operation to be done)
  - Buffered operand values (when available)
  - Reservation station number of instruction providing the operand values
- RS fetches and buffers an operand as soon as it becomes available (not necessarily involving register file)
- Pending instructions designate the RS that will provide input
- Result values broadcast on common data bus (CDB)



Dynamic Scheduling – Thomasulo's Algorithm

- Load and store buffers contain data and addresses.
- They also act like RS



Dynamic Scheduling – Thomasulo's Algorithm

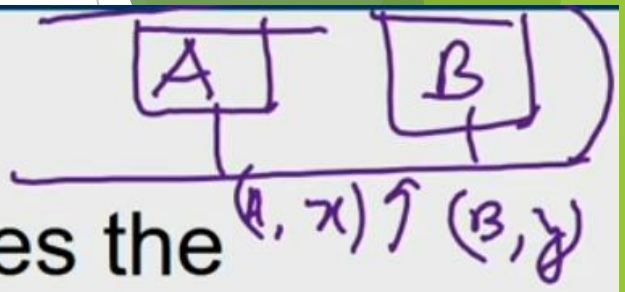
- **Issue**
  - Get next instruction from FIFO queue
  - If RS available, issue the instruction to the RS with operand values if available
  - If operand values not available, stall the instruction

Add R2, R3, R1

Dynamic Scheduling – Thomasulo's Algorithm

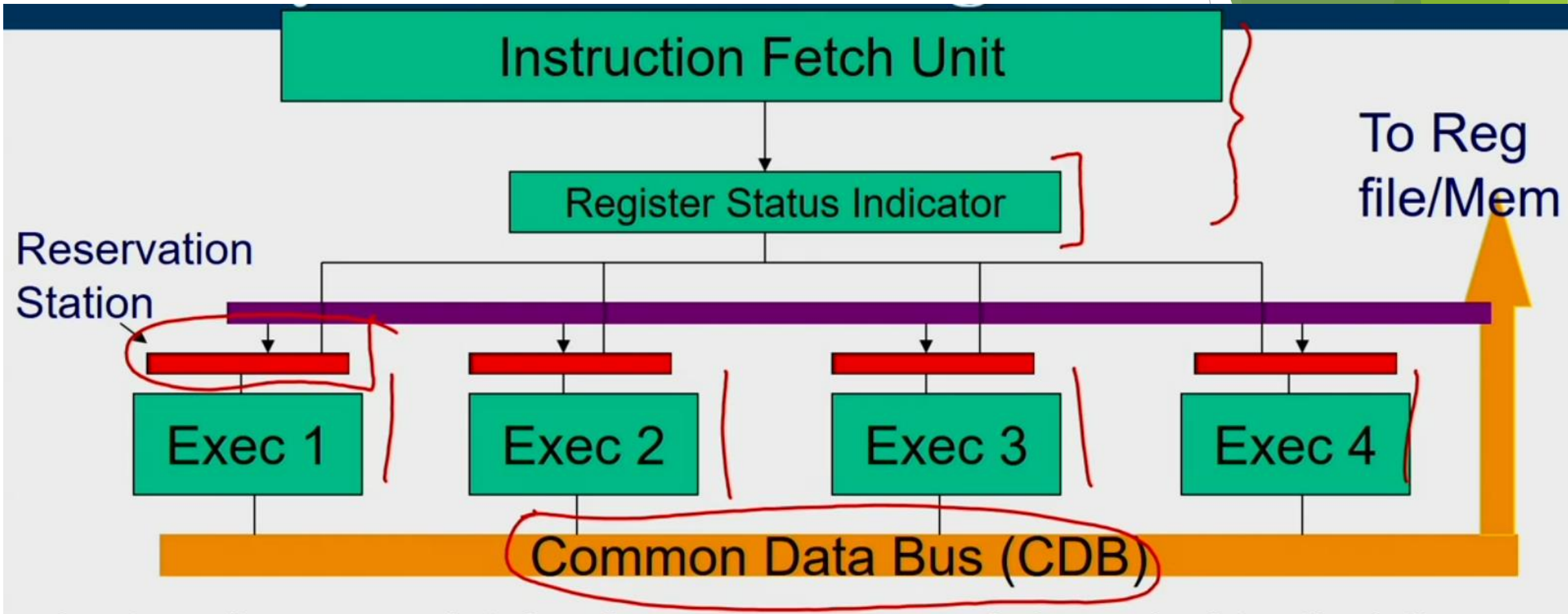
- **Execute**
  - When operand becomes available, store it in any reservation stations waiting for it
  - When all operands are ready, execute the instruction
  - Loads and store uses buffers
  - No instruction will initiate execution until all branches that precede it in program order have completed

Dynamic Scheduling – Thomasulo's Algorithm



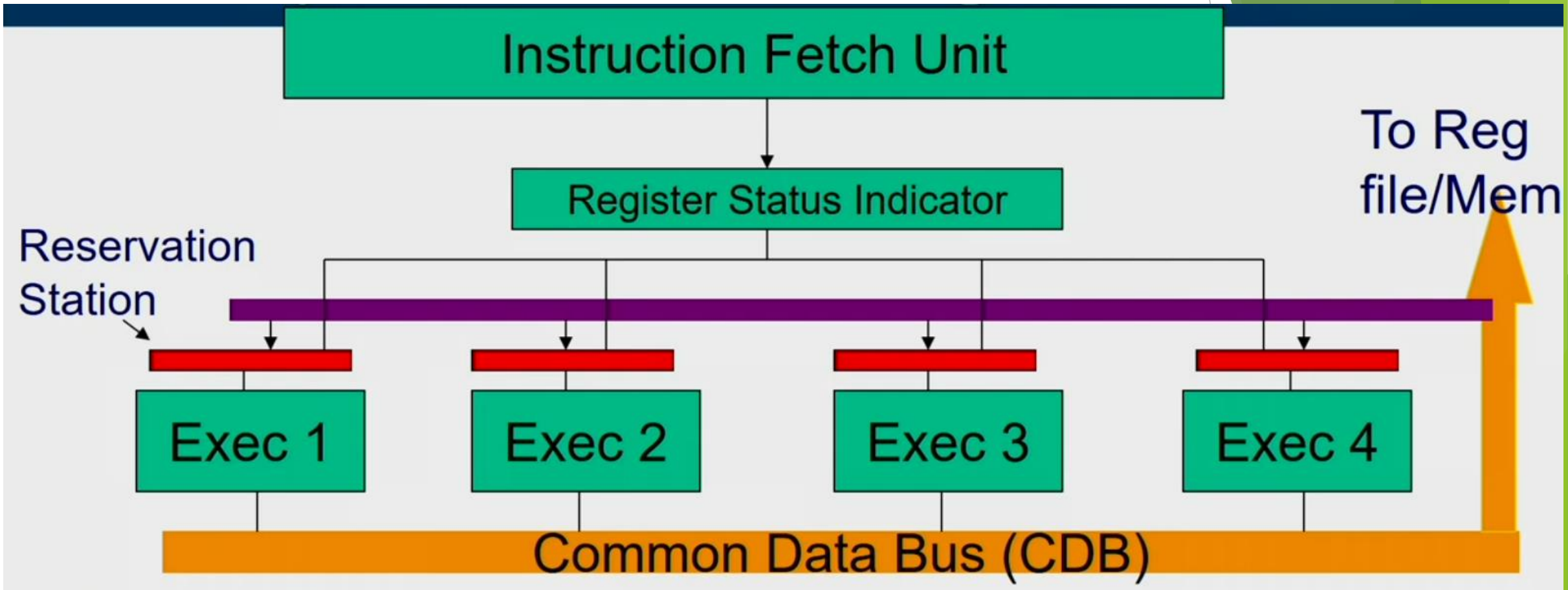
- **Write result**
  - Write result into CDB (there by it reaches the reservation station, store buffer and registers file) with name of execution unit that generated the result.
  - Stores must wait until address and value are received

Dynamic Scheduling – Thomasulo's Algorithm



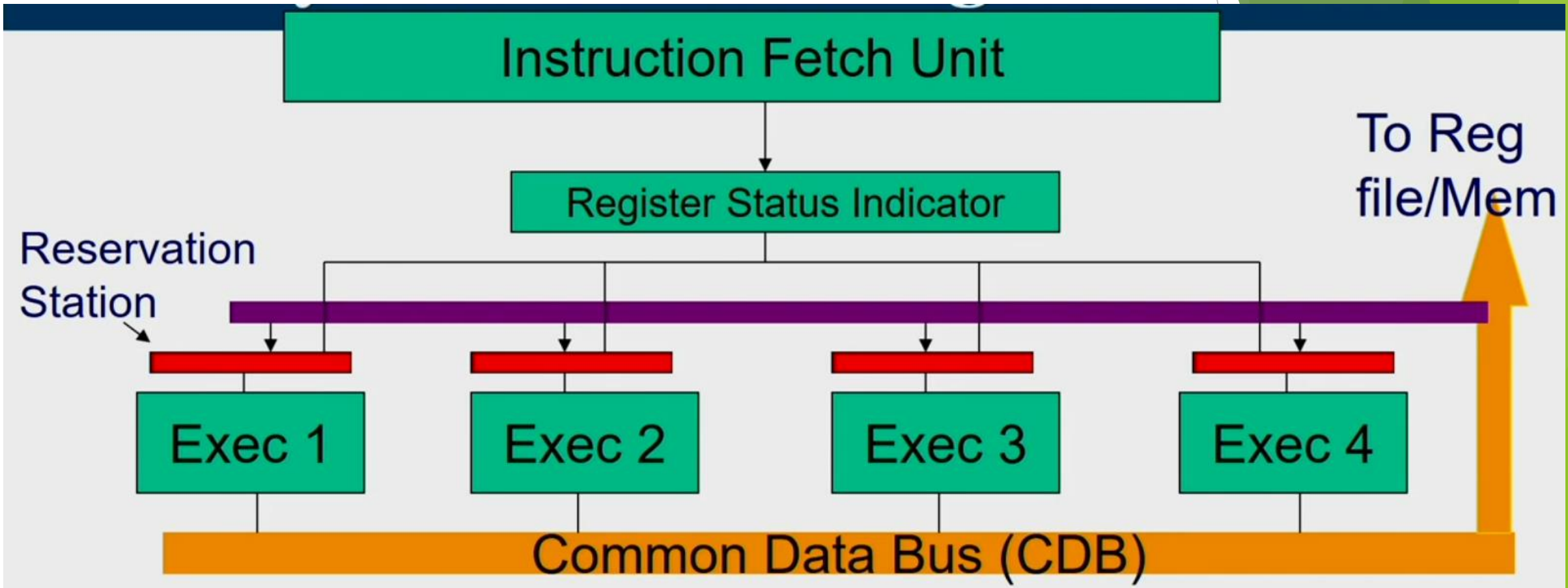
Instructions are fetched one by one and decoded to find the type of operation and the source of operands

Dynamic Scheduling – Thomasulo's Algorithm



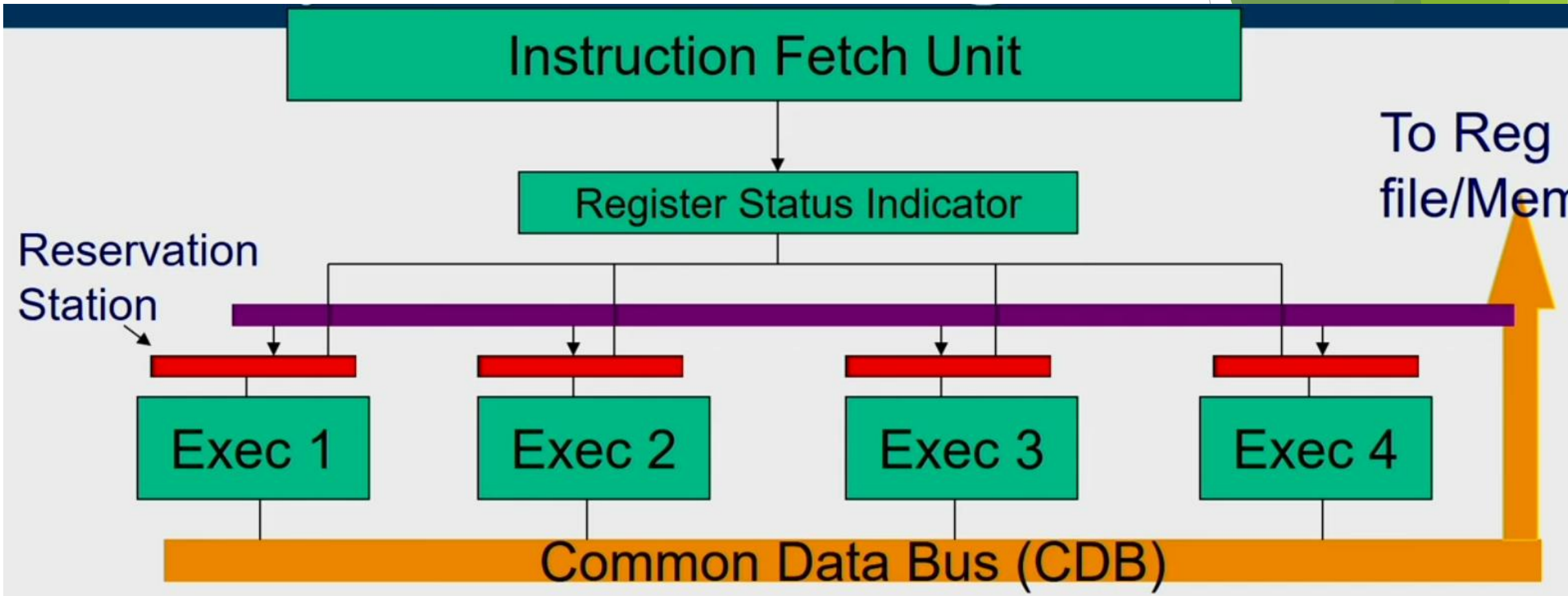
Register Status Indicator indicates whether the latest value of the register is in the reg file or currently being computed by some execution unit and if so, it states the execution unit number

Dynamic Scheduling – Thomasulo's Algorithm



If all operands available then operation proceeds in the allotted execution unit, else, it waits in the reservation station of the allotted execution unit pinging the CDB

Dynamic Scheduling – Thomasulo's Algorithm



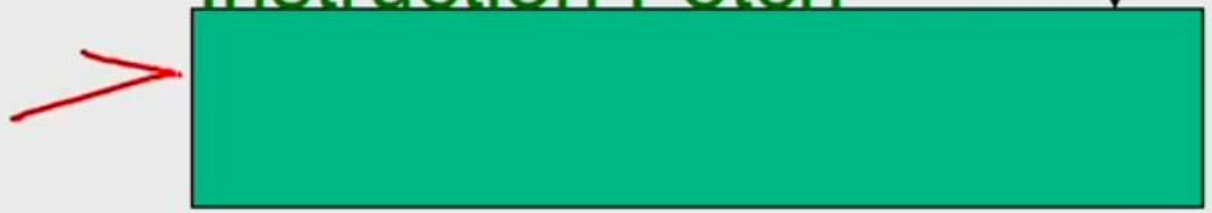
Every Execution unit writes the result along with the unit number on to the CDB which is forwarded to all reservation stations, Reg-file and Memory

Dynamic Scheduling – Thomasulo's Algorithm

An Example:

1. ADD R1, R2, R3
2. ST R1, [R4+50]
3. ADD R1, R5, R6
4. SUB R7,R1,R8
5. ST R1, [R4 + 54]
6. ADD R1, R9, R10

Instruction Fetch



Register Status Indicator

Reg Number	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Status	0	0	0	0	0	0	0	0	0	0

Empty	Empty	Empty	Empty	Empty	Empty

Dynamic Scheduling – Thomasulo's Algorithm

An Example:

1. --
2. ST R1, [R4+50]
3. ADD R1, R5, R6
4. SUB R7,R1,R8
5. ST R1, [R4 + 54]
6. ADD R1, R9, R10

Instruction Fetch

ADD R1, R2, R3

Register Status Indicator

Reg Number	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Status	1	0	0	0	0	0	0	0	0	0

Ins 1	Empty	Empty	Empty	Empty	Empty

Dynamic Scheduling – Thomasulo's Algorithm

An Example:

1. ---
2. ---
3. ADD R1, R5, R6
4. SUB R7,R1,R8
5. ST R1, [R4 + 54]
6. ADD R1, R9, R10

Instruction Fetch

ST R1, [R4+50]

Register Status Indicator

Reg Number	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Status	1	0	0	0	0	0	0	0	0	0

I 1, E	I 2, W 1	Empty	Empty	Empty	Empty

Dynamic Scheduling – Thomasulo's Algorithm

An Example:

1. ---
2. ---
3. ---
4. SUB R7,R1,R8
5. ST R1, [R4 + 54]
6. ADD R1, R9, R10

Instruction Fetch

ADD R1, R5, R6

Register Status Indicator

Reg Number	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Status	3	0	0	0	0	0	0	0	0	0

I 1, E	I 2, W 1	I 3, E	Empty	Empty	Empty

Note: Reservation Station stores the number of the execution unit that shall yield the latest value of a register.

Dynamic Scheduling – Thomasulo's Algorithm

An Example:

1. ---
2. ---
3. ---
4. ---
5. ST R1, [R4 + 54]
6. ADD R1, R9, R10

Instruction Fetch

SUB R7,R1,R8

Register Status Indicator

Reg Number	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Status	3	0	0	0	0	0	4	0	0	0

I 1, E	I 2, W 1	I 3, E	I 4, W 3	Empty	Empty

Dynamic Scheduling – Thomasulo's Algorithm

An Example:

Instruction Fetch

ST R1, [R4 + 54]

1. ---
2. ---
3. ---
4. ---
5. ---
6. ADD R1, R9, R10

Register Status Indicator

Reg Number	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Status	3	0	0	0	0	0	4	0	0	0

I 1, E	I 2, W 1	I 3, E	I 4, W 3	I 5, W 3	Empty

Dynamic Scheduling – Thomasulo's Algorithm

An Example:

1. ADD R1, R2, R3
2. ST U1, [R4+50]
3. ADD R1, R5, R6
4. SUB R7, U3, R8
5. ST U3, [R4 + 54]
6. ADD R1, R9, R10

Instruction Fetch

ADD R1, R9, R10

Register Status Indicator

Reg Number	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Status	6	0	0	0	0	0	4	0	0	0

I 1, E | I 2, W 1 | I 3, E | I 4, W 3 | I 5, W 3 | I 6, E

Effectively three Instructions are executing and others waiting for the appropriate results. The whole program is converted as shown above.

Dynamic Scheduling – Thomasulo's Algorithm

An Example:

1. ADD R1, R2, R3
2. ST U1, [R4+50]
3. ADD R1, R5, R6
4. SUB R7, U3, R8
5. ST U3, [R4 + 54]
6. ADD R1, R9, R10

WAW

Instruction Fetch

ADD R1, R9, R10

Register Status Indicator

Reg Number	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Status	6	0	0	0	0	0	4	0	0	0

I 1, E	I 2, W 1	I 3, E	I 4, W 3	I 5, W 3	I 6, E

Operand Forwarding and Register Renaming is done automatically

Dynamic Scheduling – Thomasulo's Algorithm

An Example:

1. ADD R1, R2, R3
2. ST U1, [R4+50]
3. ADD R1, R5, R6
4. SUB R7, U3, R8
5. ST U3, [R4 + 54]
6. ADD R1, R9, R10

Instruction Fetch

ADD R1, R9, R10

Register Status Indicator

Reg Number	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Status	6	0	0	0	0	0	4	0	0	0

I 1, E	I 2, W 1	I 3, E	I 4, W 3	I 5, W 3	I 6, E

Execution unit 6, on completion will make R1 entry in Register Status Indicator 0. Similarly unit 4 will make R7 entry 0.

Dynamic Scheduling – Thomasulo's Algorithm

- **Each reservation station has seven fields.**
- **{Op, Qj, Qk, Vj, Vk, A, Busy}**

1. **Op** —The operation to perform on source operands S1 and S2.

2,3. **Qj, Qk** —The reservation stations that will produce the corresponding source operand; a value of zero indicates that the source operand is already available in Vj or Vk.

4,5. **Vj, Vk** —The value of the source operands.

Only one of the **V field** or the **Q field** is valid for each operand. For loads, the Vk field is used to hold the offset field.

Dynamic Scheduling – Thomasulo's Algorithm

- **Each reservation station has seven fields.**
- **{Op, Qj, Qk, Vj, Vk, A, Busy}**

6. **A** —Used to hold information for the memory address calculation for a load or store. Initially, the immediate field of the instruction is stored here; after the address calculation, the effective address is stored here.
7. **Busy** —Indicates that this reservation station and its accompanying functional unit are occupied.

Dynamic Scheduling – Thomasulo's Algorithm

- **The register file has a field, Qi: (RSI)**
- Qi—The number of the reservation station that contains the operation whose result should be stored into this register.
- If the value of Qi = 0 no currently active instruction is computing a result destined for this register, meaning that the value is simply the register contents.