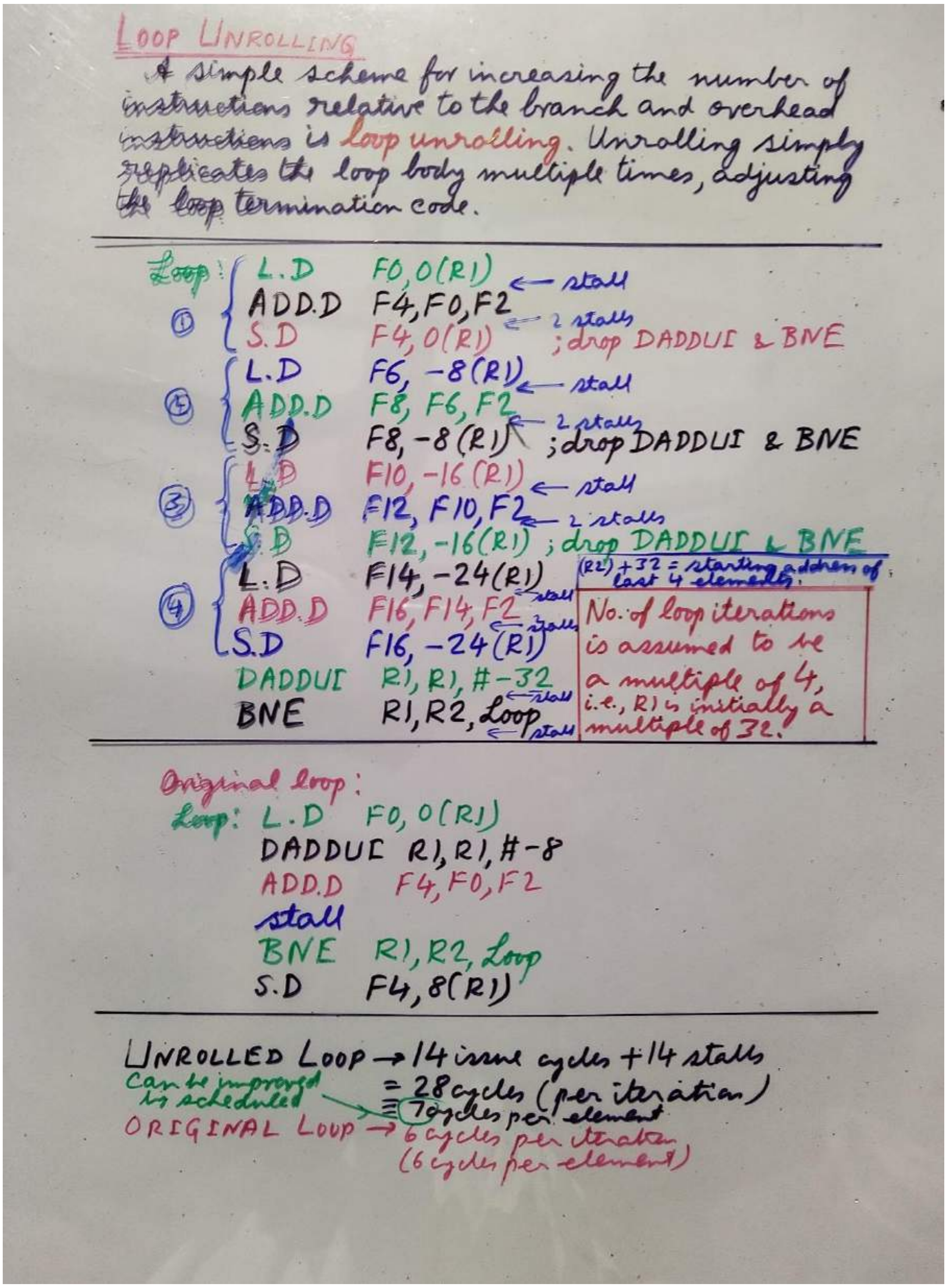
Notes-09

LOOP Unrolling

Informal: A loop is typically implemented by having a loop termination code at the beginning or end of the loop. This code is really an overhead; it doesn’t participate in the computation involved in the loop iterations. For a large number of iterations this overhead can be significant. **Unrolling** the loop, i.e. replicating the loop body multiple number of times instead of running the loop termination code at the end of each iteration can reduce this overhead. Of course, it is not practical to do this to an arbitrary extent. Nevertheless, it **CAN** reduce execution time by a significant amount.



Loop Unrolling

A simple scheme for increasing the number of instructions relative to the branch and overhead instructions is loop unrolling. Unrolling simply replicates the loop body multiple times, adjusting the loop termination code.

```
Loop: ① { L.D     F0,0(R1)     ← stall
        { ADD.D   F4,F0,F2     ← 2 stalls
        { S.D     F4,0(R1)     ;drop DADDUI & BNE
      ② { L.D     F6,-8(R1)    ← stall
        { ADD.D   F8,F6,F2     ← 2 stalls
        { S.D     F8,-8(R1)    ;drop DADDUI & BNE
      ③ { L.D     F10,-16(R1)  ← stall
        { ADD.D   F12,F10,F2   ← 2 stalls
        { S.D     F12,-16(R1)  ;drop DADDUI & BNE
      ④ { L.D     F14,-24(R1)  ← stall
        { ADD.D   F16,F14,F2   ← 2 stalls
        { S.D     F16,-24(R1)
          DADDUI  R1,R1,#-32   ← stall
          BNE     R1,R2,Loop   ← stall
```

(R2)+32 = starting address of last 4 elements.

No. of loop iterations is assumed to be a multiple of 4, i.e., R1 is initially a multiple of 32.

Original loop:

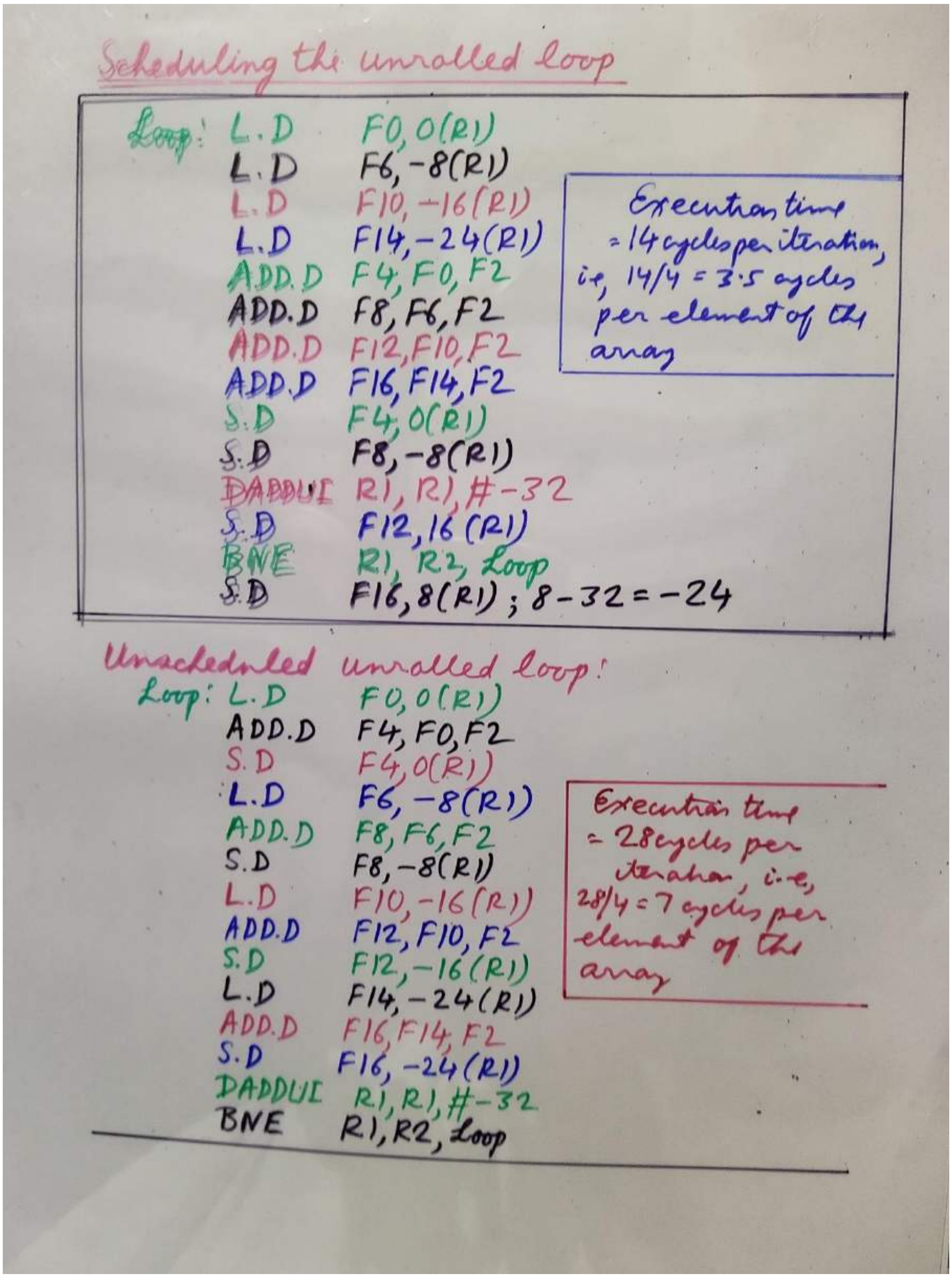
```
Loop: L.D     F0,0(R1)
      DADDUI  R1,R1,#-8
      ADD.D   F4,F0,F2
      stall
      BNE     R1,R2,Loop
      S.D     F4,8(R1)
```

UNROLLED LOOP → 14 issue cycles + 14 stalls
= 28 cycles (per iteration)
≡ 7 cycles per element

Can be improved is scheduled

ORIGINAL LOOP → 6 cycles per iteration
(6 cycles per element)

Slide 9.1



Scheduling the unrolled loop

```
Loop: 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     F4,0(R1)
      S.D     F8,-8(R1)
      DADDUI  R1,R1,#-32
      S.D     F12,16(R1)
      BNE     R1,R2,Loop
      S.D     F16,8(R1) ; 8-32=-24
```

Execution time = 14 cycles per iteration, i.e, 14/4 = 3.5 cycles per element of the array

Unscheduled unrolled loop:

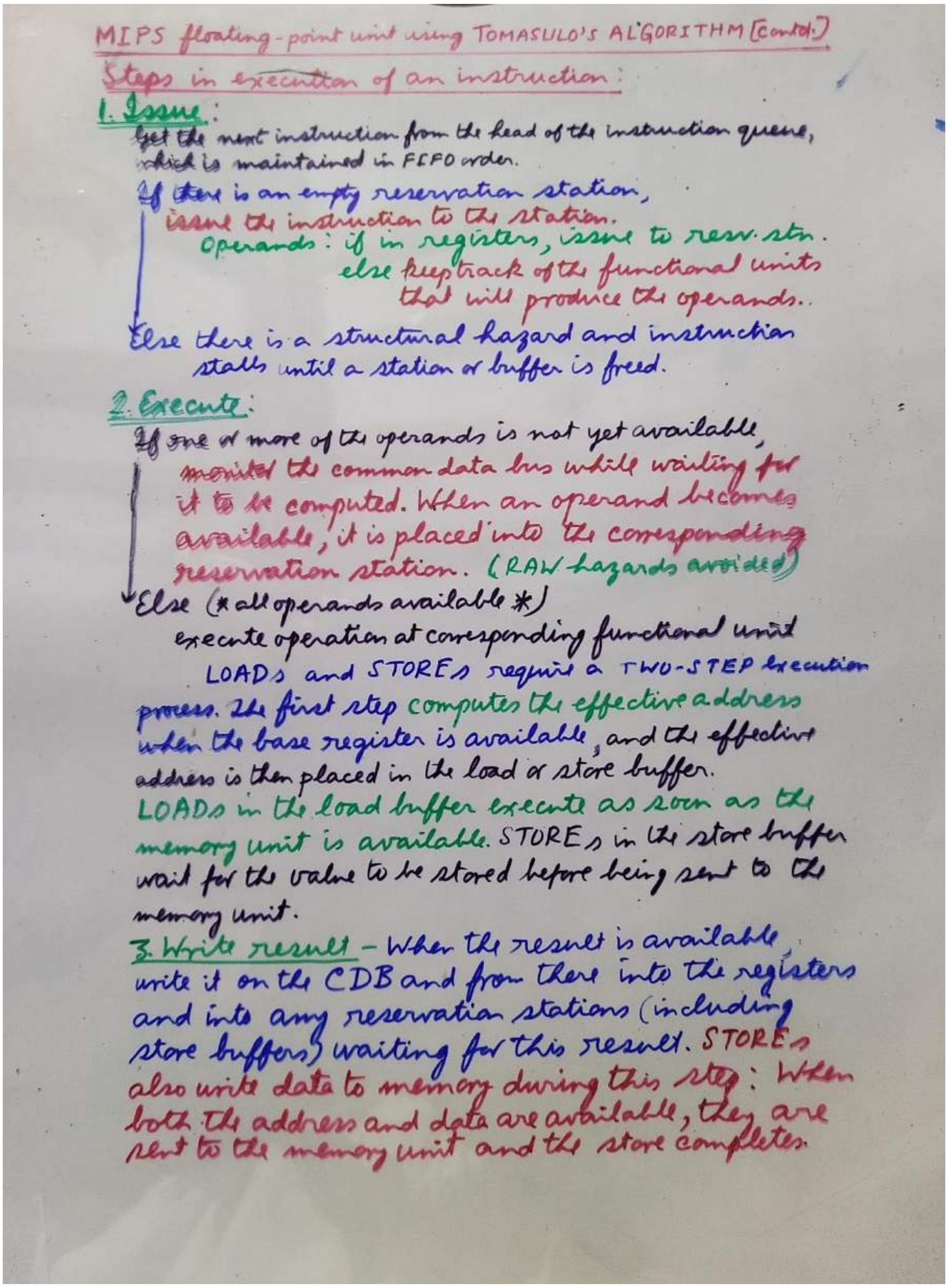
```
Loop: L.D     F0,0(R1)
      ADD.D   F4,F0,F2
      S.D     F4,0(R1)
      L.D     F6,-8(R1)
      ADD.D   F8,F6,F2
      S.D     F8,-8(R1)
      L.D     F10,-16(R1)
      ADD.D   F12,F10,F2
      S.D     F12,-16(R1)
      L.D     F14,-24(R1)
      ADD.D   F16,F14,F2
      S.D     F16,-24(R1)
  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

Execution time = 28 cycles per iteration, i.e, 28/4 = 7 cycles per element of the array

Slide 9.2

DYNAMIC SCHEDULING

Informal: Performance can be improved to a significant extent if we take a radically different viewpoint regarding instruction execution. The traditional approach is to take an instruction, execute it, and then pick up the next instruction. Pipelining also takes this approach, although it initiates the next instruction before completion of the current instruction. In DYNAMIC SCHEDULING, an instruction is scheduled to execute dynamically, i.e. at run time, depending upon the availability of its input operands. We don't view it as sequential execution. Thus, we can view the program as a set of instructions, each waiting eagerly for its input operands to arrive. As soon as the operands arrive, the instruction executes, subject, of course, to the availability of functional units.



MIPS floating-point unit using TOMASULO'S ALGORITHM [Contd.]

Steps in execution of an instruction:

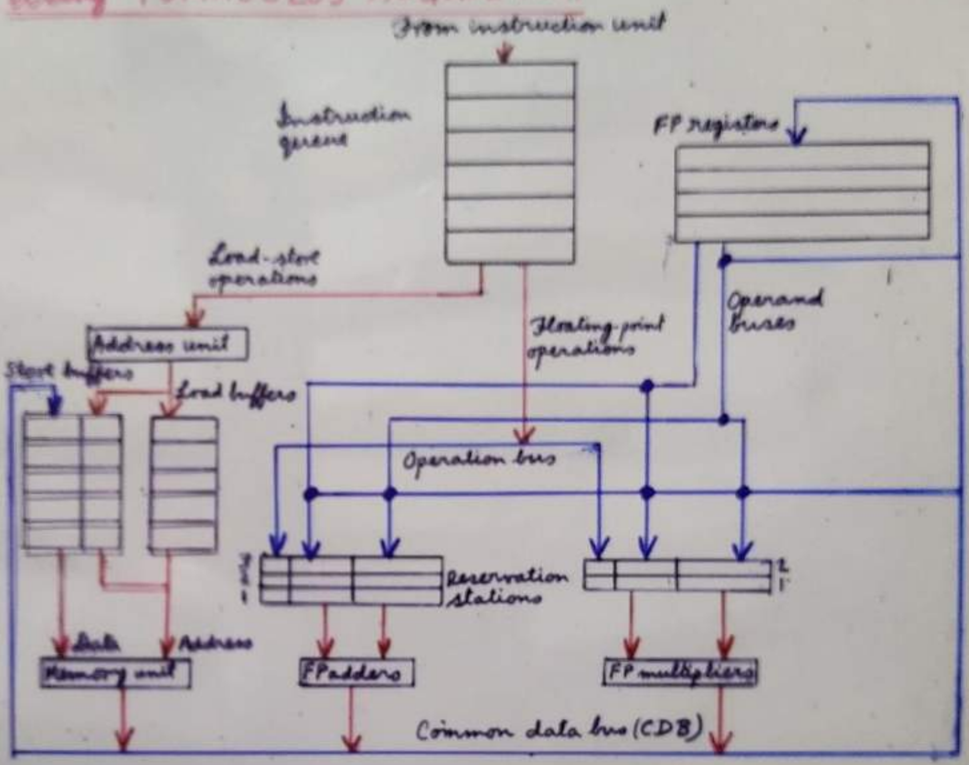
1. Issue:
Get the next instruction from the head of the instruction queue, which is maintained in FIFO order.
If there is an empty reservation station,
issue the instruction to the station.
Operands: if in registers, issue to resv. stn.
else keep track of the functional units that will produce the operands.
Else there is a structural hazard and instruction stalls until a station or buffer is freed.

2. Execute:
If one or more of the operands is not yet available,
monitor the common data bus while waiting for it to be computed. When an operand becomes available, it is placed into the corresponding reservation station. (RAW hazards avoided)
Else (*all operands available*)
execute operation at corresponding functional unit
LOADs and STOREs require a TWO-STEP execution process. The first step computes the effective address when the base register is available, and the effective address is then placed in the load or store buffer.
LOADs in the load buffer execute as soon as the memory unit is available. STOREs in the store buffer wait for the value to be stored before being sent to the memory unit.

3. Write result - When the result is available, write it on the CDB and from there into the registers and into any reservation stations (including store buffers) waiting for this result. STOREs also write data to memory during this step: When both the address and data are available, they are sent to the memory unit and the store completes.

Slide 9.3

Dynamic Scheduling: MIPS FLOATING-POINT unit using TOMASULO'S ALGORITHM



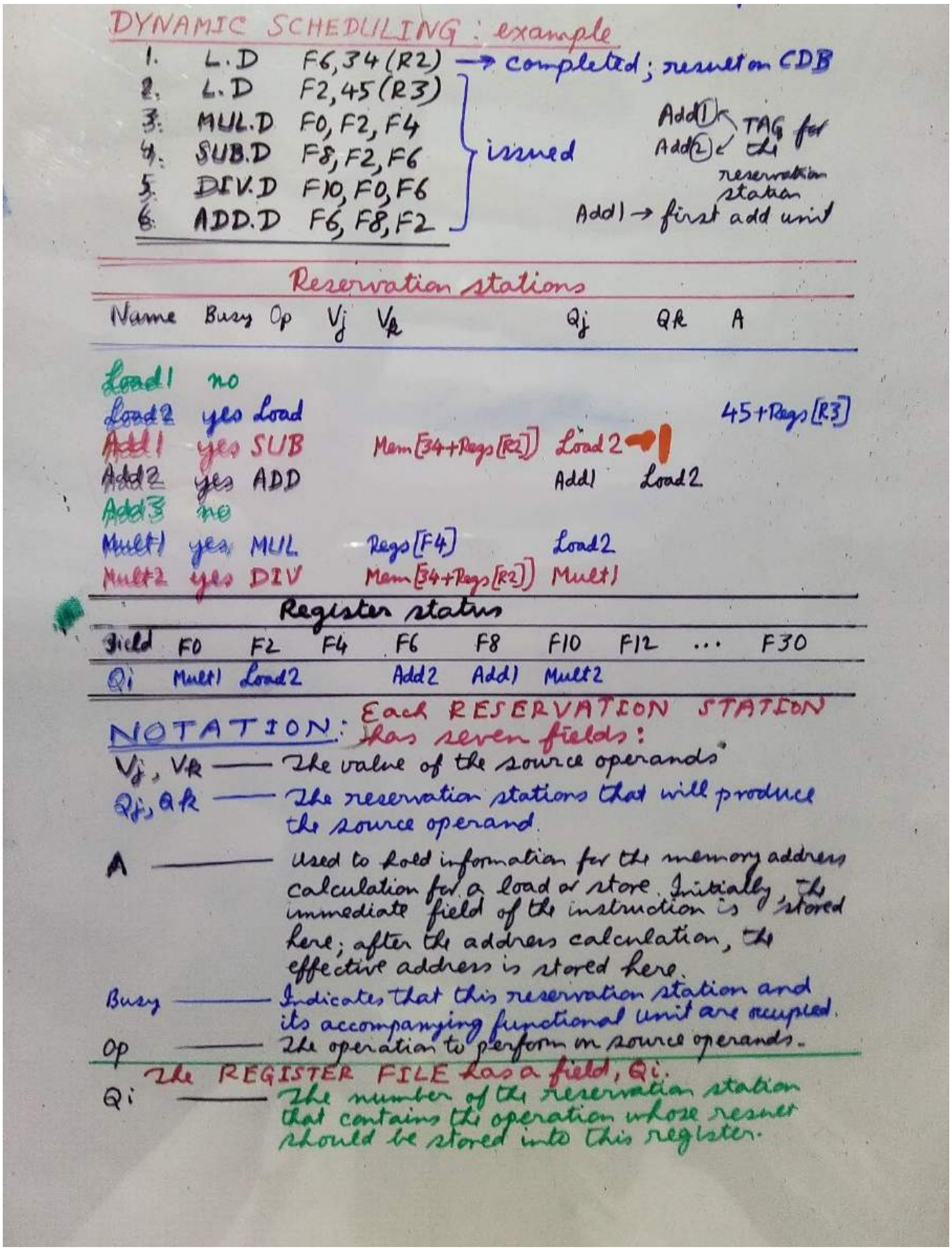
Each reservation station holds an instruction that has been issued and is awaiting execution at a functional unit. It also contains the operand values for that instruction (if they have already been computed) OR the names of the reservation stations that will provide the operand values.

Load buffers (i) hold the components of the effective address until it is computed; (ii) track outstanding loads that are waiting on the memory; (iii) hold the results of completed loads that are waiting for the CDB.

Store buffers (i) hold the components of the effective address (ii) hold addresses of outstanding stores (iii) hold address and value until memory is available.

All results from the functional units and from memory are sent on the common data bus, which goes everywhere except to the load buffer.

Slide 9.4



DYNAMIC SCHEDULING : example

```
1. L.D    F6,34(R2)  → completed; result on CDB
2. L.D    F2,45(R3)  ┐
3. MUL.D  F0,F2,F4   │
4. SUB.D  F8,F2,F6   │ issued
5. DIV.D  F10,F0,F6  │
6. ADD.D  F6,F8,F2   ┘
```

Add1, Add2 — TAG for the reservation station

Add1 → first add unit

Reservation stations

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	no						
Load2	yes	Load					45+Regs[R3]
Add1	yes	SUB		Mem[34+Regs[R2]]	Load2		
Add2	yes	ADD			Add1	Load2	
Add3	no						
Mult1	yes	MUL		Regs[F4]	Load2		
Mult2	yes	DIV		Mem[34+Regs[R2]]	Mult1		

Register status

Field	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	Mult1	Load2		Add2	Add1	Mult2			

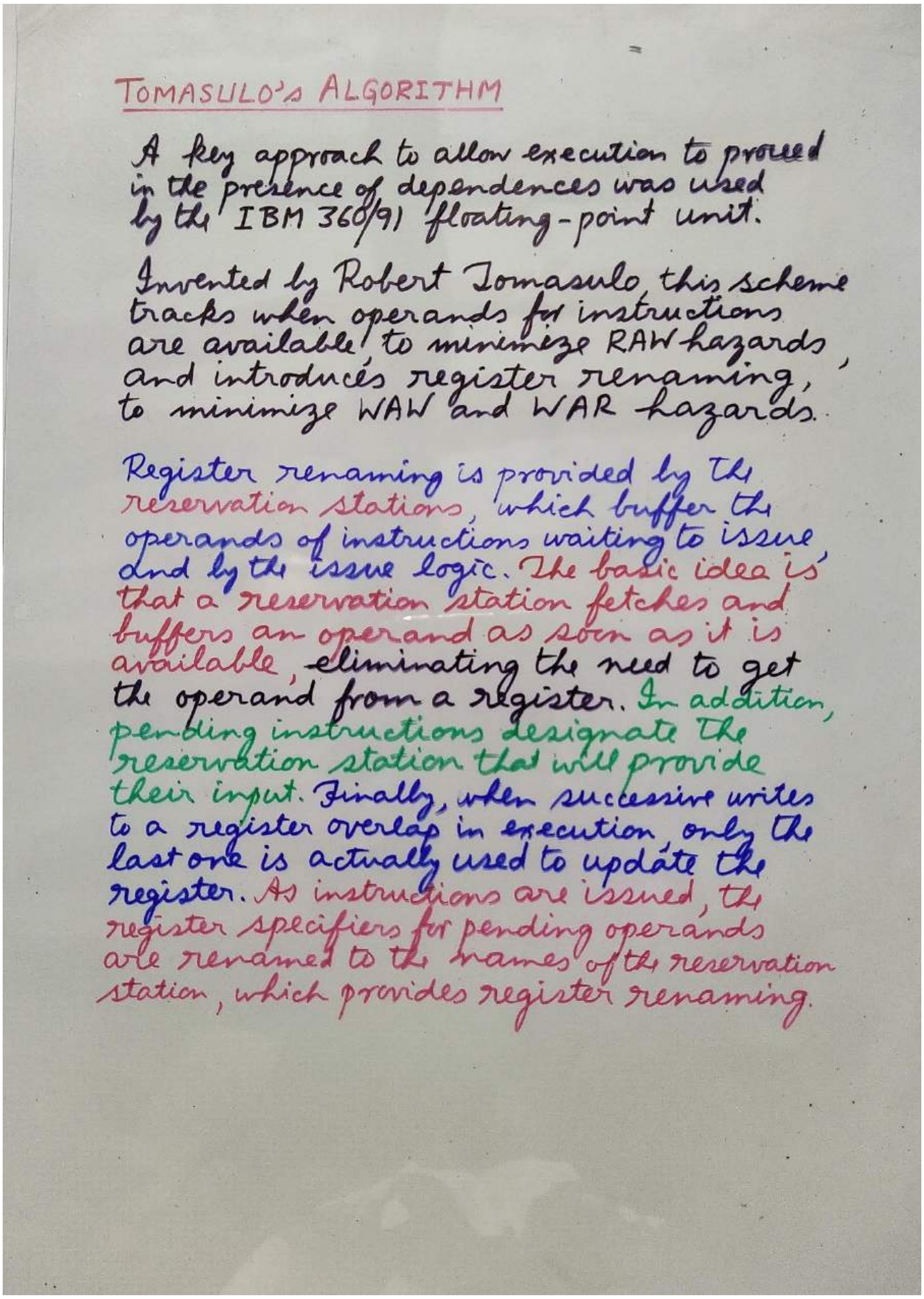
NOTATION: Each RESERVATION STATION has seven fields:

Vj, Vk	The value of the source operands
Qj, Qk	The reservation stations that will produce the source operand.
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.
Busy	Indicates that this reservation station and its accompanying functional unit are occupied.
Op	The operation to perform on source operands.

The REGISTER FILE has a field, Qi.

Qi	The number of the reservation station that contains the operation whose result should be stored into this register.

Slide 9.5



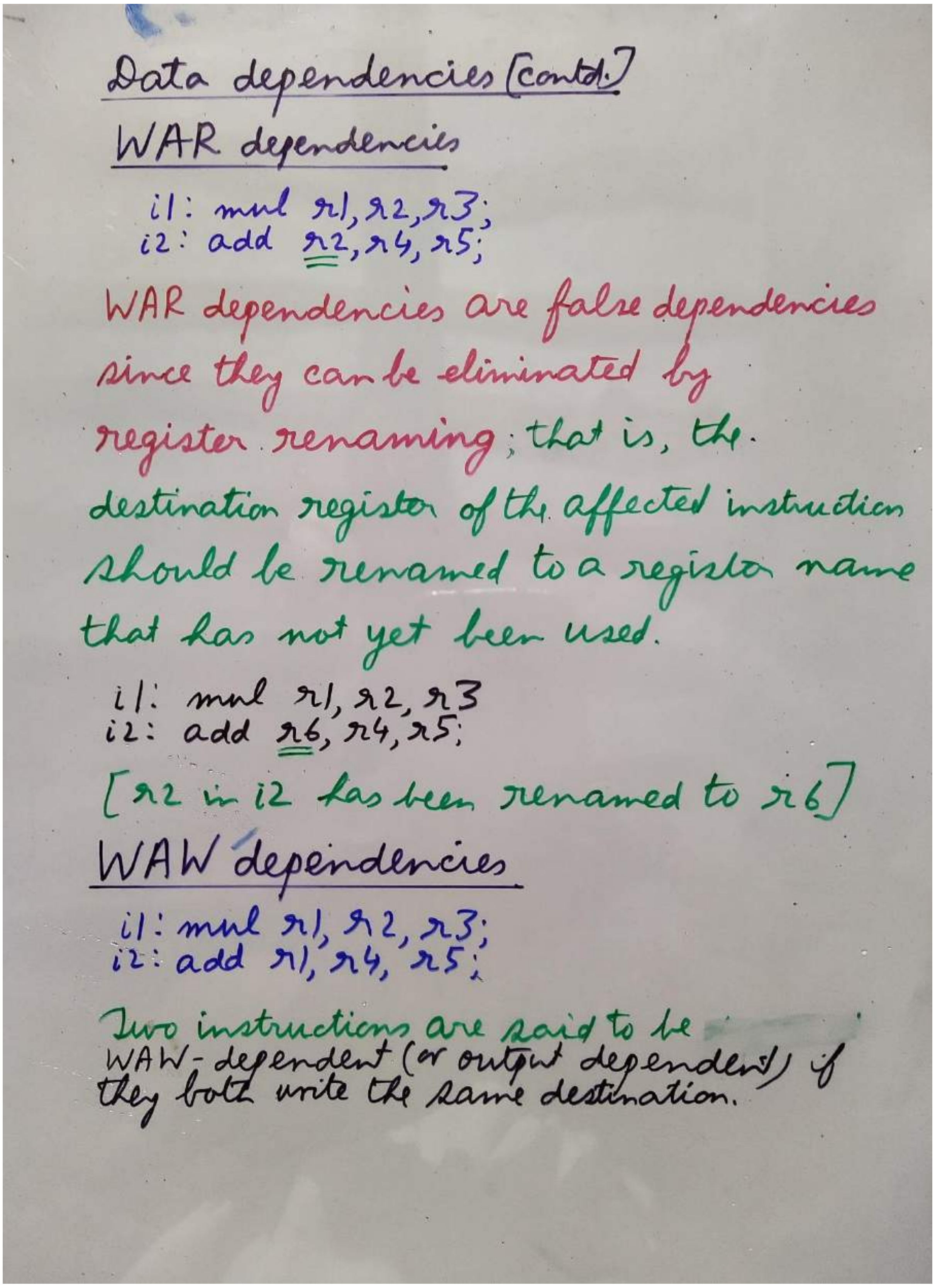
TOMASULO's ALGORITHM

A key approach to allow execution to proceed in the presence of dependences was used by the IBM 360/91 floating-point unit.

Invented by Robert Tomasulo, this scheme tracks when operands for instructions are available, to minimize RAW hazards, and introduces register renaming, to minimize WAW and WAR hazards.

Register renaming is provided by the reservation stations, which buffer the operands of instructions waiting to issue, and by the issue logic. The basic idea is that a reservation station fetches and buffers an operand as soon as it is available, eliminating the need to get the operand from a register. In addition, pending instructions designate the reservation station that will provide their input. Finally, when successive writes to a register overlap in execution, only the last one is actually used to update the register. As instructions are issued, the register specifiers for pending operands are renamed to the names of the reservation station, which provides register renaming.

Slide 9.6



Data dependencies [contd.]

WAR dependencies

```
i1: mul r1, r2, r3;
i2: add r2, r4, r5;
```

WAR dependencies are false dependencies since they can be eliminated by register renaming; that is, the destination register of the affected instruction should be renamed to a register name that has not yet been used.

```
i1: mul r1, r2, r3
i2: add r6, r4, r5;
```

[r2 in i2 has been renamed to r6]

WAW dependencies

```
i1: mul r1, r2, r3;
i2: add r1, r4, r5;
```

Two instructions are said to be WAW-dependent (or output dependent) if they both write the same destination.

Slide 9.7