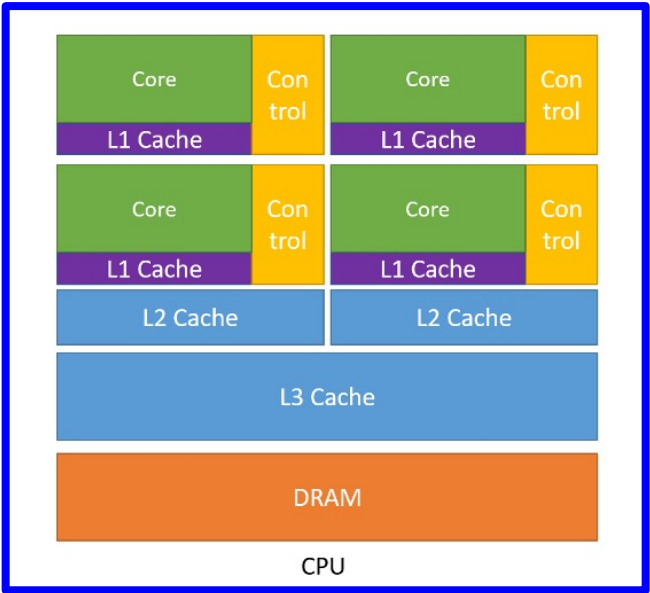
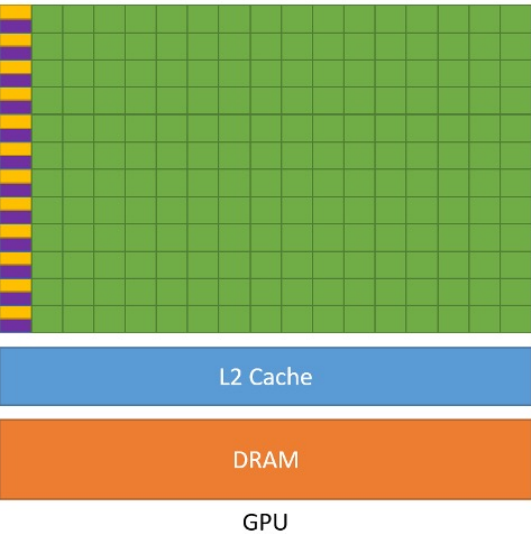
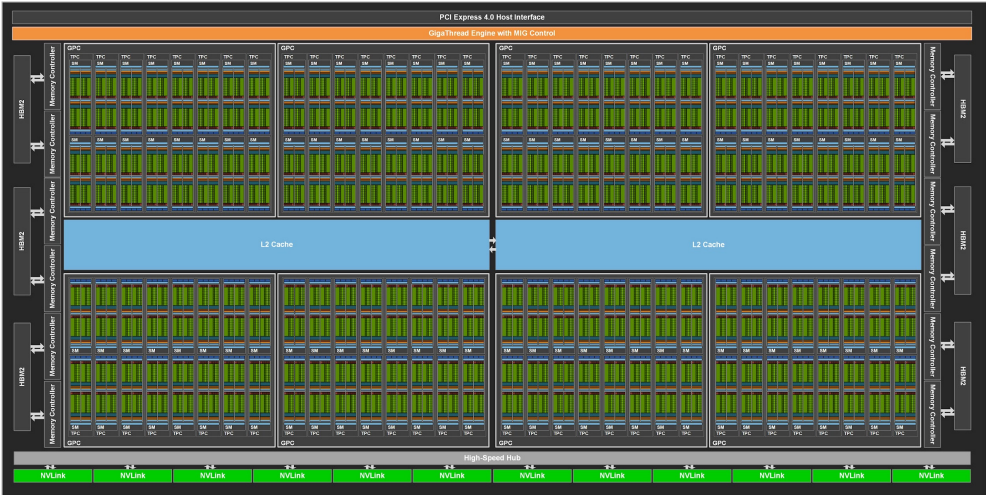


# So far..



# **Multicore Architecture (Part1)**

Prof. Seokin Hong

# Agenda

---

- **Instruction-Level Parallelism (ILP) Concepts and Challenge**
  - Instruction-level Parallelism Concepts
  - Compiler Techniques for Exposing ILP
  - Dynamic Scheduling for Overcoming Data Hazards
  - Limitations of ILP
  
- Thread-Level Parallelism (TLP)
  - Why TLP?
  - Simultaneous Multi-threading (SMT)
  - Multi-core Architecture

---

# **Instruction-Level Parallelism**

## Concepts and Challenges

# ILP Basics

---

## ■ Instruction level parallelism (ILP)

- Overlaps execution of instructions

## ■ Goal: minimize CPI (maximize IPC)

|                  |                   |  |
|------------------|-------------------|--|
| Loop: <b>L.D</b> | <b>F1,0(R1)</b>   | <br>Execute these<br>instruction in parallel |
| <b>ADD.D</b>     | <b>F4,F0,F2</b>   |  |
| <b>ADD.D</b>     | <b>F5,F0,F2</b>   |  |
| <b>ADD.D</b>     | <b>F6,F0,F2</b>   |  |
| <b>ADD.D</b>     | <b>F7,F0,F2</b>   |  |
| <b>S.D</b>       | <b>F8,0(R1)</b>   |  |
| <b>DADDUI</b>    | <b>R1,R1,#-8</b>  |  |
| <b>BNE</b>       | <b>R1,R2,LOOP</b> |  |

# Challenges

---

- **Challenges:** Not all instructions can be executed in parallel
- In a pipelined processor
  - $\text{CPI} = \text{ideal CPI} + (\text{Structural stalls} + \text{Data hazard stalls} + \text{Control stalls})$
- Dependent instructions cannot be executed simultaneously
  - **Three types of dependences**
    - Data dependences
    - Name dependences
    - Control dependences

# Data Dependence : Basics

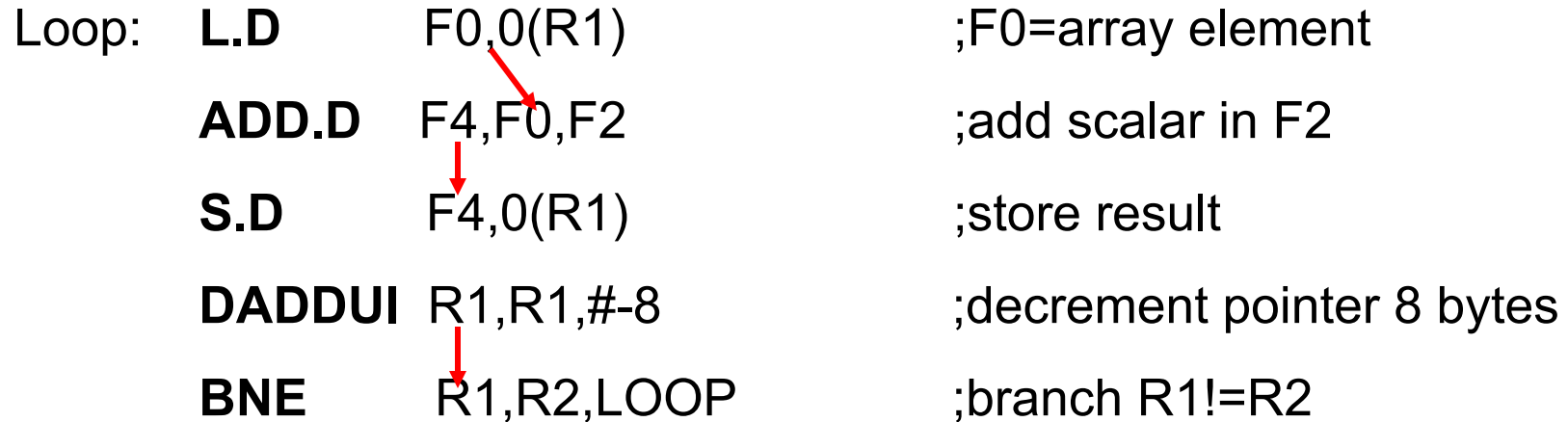
---

- Instruction j is **data dependent** on instruction i if instruction i produces results used by instruction j
- Data dependent instructions have to be executed “in order”
- **Pipeline interlocks**
  - With interlocks, data dependence causes a hazard and stall
  - Without interlocks, data dependence prohibits the compiler from scheduling instructions with overlap
- Data dependence conveys:
  - Possibility of a hazard
  - The required order of instructions
  - Upper bound on achievable parallelism

# Data Dependence : Example

---

Loop:   **L.D**       F0,0(R1)                   ;F0=array element  
          **ADD.D**   F4,F0,F2                   ;add scalar in F2  
          **S.D**       F4,0(R1)                  ;store result  
          **DADDUI** R1,R1,#-8                  ;decrement pointer 8 bytes  
          **BNE**       R1,R2,LOOP               ;branch R1!=R2





# Data Dependence : Details

---


- Data dependence can limit the amount of instruction-level parallelism
- **Overcoming data dependence**
  - Maintain the dependence by preventing the hazard by compiler or by hardware scheduler
  - Eliminate dependence by transforming the code
- Detecting the data dependence is straightforward since the register names are fixed in the instructions
- But, data dependences that flow through memory locations are difficult to detect, since two addresses may refer to the same location but look different:
  - Is 100(R4) the same as 20(R6) ?

# Name Dependence

---


- **Name dependence** occurs when two instructions use the same register (or memory location), called name, but there is no flow of data between them
- **Two types of name dependence**
  - **Antidependence:** instruction *j* writes a register (or memory location) that instruction *i* reads.

```
Loop:  L.D      F0,0(R1)
        ADD.D   F4,F0,F2
        S.D     F4,0(R1)      ;instruction i
        DADDUI  R1,R1,#-8     ;instruction j
        BNE     R1,R2,LOOP
```



- **Output dependence:** instruction *i* and *j* write the same register (or memory location)

```
Loop:  L.D      F0,0(R1)
        ADD.D   F4,F0,F2      ;instruction i
        ADD.D   F4,F1,F2      ;instruction j
```



# Data Hazards

---

- **Hazard** exists whenever there is a name or data dependence between instructions
  - Overlap of dependent instructions could change the outcome of the program
- The goal of both our software and hardware techniques for ILP is to exploit parallelism by preserving program order only where it affects the outcome of the program.
- Detecting and avoiding hazards ensures that necessary program order is preserved.

# Data Hazards (Cont'd)

---

- **Possible data hazards**

- **RAW (Read after write) ← true data dependence**

- Instruction i : write to x
    - Instruction j : read from x

- **WAW (Write after write) ← output dependence**

- Instruction i: write to x
    - Instruction j: write to x

- **WAR (Write after read) ← antidependence**

- Instruction i: read from x
    - Instruction j: write to x

- **RAR (Read after read) is not a hazard**

# Control Dependence

---

- Control dependence determines ordering of an instruction with respect to **a branch instruction**
  - The order must be preserved
  - The execution should be conditional

- **Example**

```
if p1 {  
    S1;    // S1 is control dependent on p1  
}  
if p2 {  
    S2;    // S2 is control dependent on p2  
}
```

# Control Dependence (Cont'd)

---

- Branches create barrier in code for potential ILP
- It might be possible to **violate control dependence but preserve correct execution**
  - Speculative execution

- Example

DADDU R2,R3,R4

BEQ R2, R5, L1

LW R1,0(R2)

ADD. R4, R5, R6

L1: .....

- No data dependence between BEQ (branch) and LW (load) instructions
- But, there is control dependence between these instructions

---

# **Compiler Techniques for Exposing ILP**

# Compiler Techniques for Exposing ILP

- Compiler can enhance a processor's ability to exploit ILP

- Example:

for (i=999; i>=0; i=i-1)

    x[i] = x[i] + s;



```
Loop:    L.D      F0,0(R1)      ;F0=array element
          ADD.D    F4,F0,F2      ;add scalar in F2
          S.D      F4,0(R1)      ;store result
          DADDUI   R1,R1,#-8     ;decrement pointer
                                   ;8 bytes (per DW)
          BNE      R1,R2,Loop    ;branch R1!=R2
```

| Instruction producing result | Instruction using result | Latency in clock cycles |
|------------------------------|--------------------------|-------------------------|
| FP ALU op                    | Another FP ALU op        | 3                       |
| FP ALU op                    | Store double             | 2                       |
| Load double                  | FP ALU op                | 1                       |
| Load double                  | Store double             | 0                       |



# Instruction Scheduling Example

---

- Without any scheduling

Loop: L.D F0,0(R1)

stall

ADD.D F4,F0,F2

stall

stall

S.D. F4,0(R1)

DADDUI R1,R1,#-8

stall

BNE R1,R2,Loop

**Takes 9 clock cycles**

| Instruction producing result | Instruction using result | Latency in clock cycles |
|------------------------------|--------------------------|-------------------------|
| FP ALU op                    | Another FP ALU op        | 3                       |
| FP ALU op                    | Store double             | 2                       |
| Load double                  | FP ALU op                | 1                       |
| Load double                  | Store double             | 0                       |

# Instruction Scheduling Example

---

- With scheduling

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

**Takes 6 clock cycles**

| Instruction producing result | Instruction using result | Latency in clock cycles |
|------------------------------|--------------------------|-------------------------|
| FP ALU op                    | Another FP ALU op        | 3                       |
| FP ALU op                    | Store double             | 2                       |
| Load double                  | FP ALU op                | 1                       |
| Load double                  | Store double             | 0                       |

# Loop Unrolling

---

- Loop unrolling **simply copies the body of the loop multiple times**, each copy operates on a new loop
  
- Benefits
  - **Less branch instructions**
    - Less pressure on branch predictor
  - **Increased basic block size**
    - Potential for more parallelism
  - **Less instructions executed**
    - For example: less increments of the loop counter
  
- **Downsides**
  - Greater register pressure
  - Increased use of instruction cache

# Loop Unrolling : Example

---

- Unroll by a factor of 4 (assume # elements is divisible by 4)
- Eliminate unnecessary instructions

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

## Original code

```
Loop:    L.D    F0,0(R1)      ;F0=array element
          ADD.D  F4,F0,F2     ;add scalar in F2
          S.D    F4,0(R1)     ;store result
          DADDUI R1,R1,#-8     ;decrement pointer
                                   ;8 bytes (per DW)
          BNE    R1,R2,Loop    ;branch R1!=R2
```

# Loop Unrolling : Example

---

## ■ Loop Unrolling + Scheduling

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)  
          S.D F16,8(R1)  
          BNE R1,R2,Loop

### Just with loop unrolling

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

---

# Dynamic Scheduling

# Limitations of static compiler techniques

---

- A major limitation of simple pipelining techniques
  - in-order instruction issue and execution
    - → Instructions are issued in program order, and if an instruction is stalled in the pipeline no later instructions can proceed
    - → If instruction j depends on a long-running instruction i, then all instructions after j must be stalled until i is finished and j can execute

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

- Static techniques can only eliminate some data dependence stalls
  - Some dependences are not known until runtime
  - Compiler might not know the details of the micro-architecture
  - There could be unpredictable delays: multi-level caches

# Idea

---

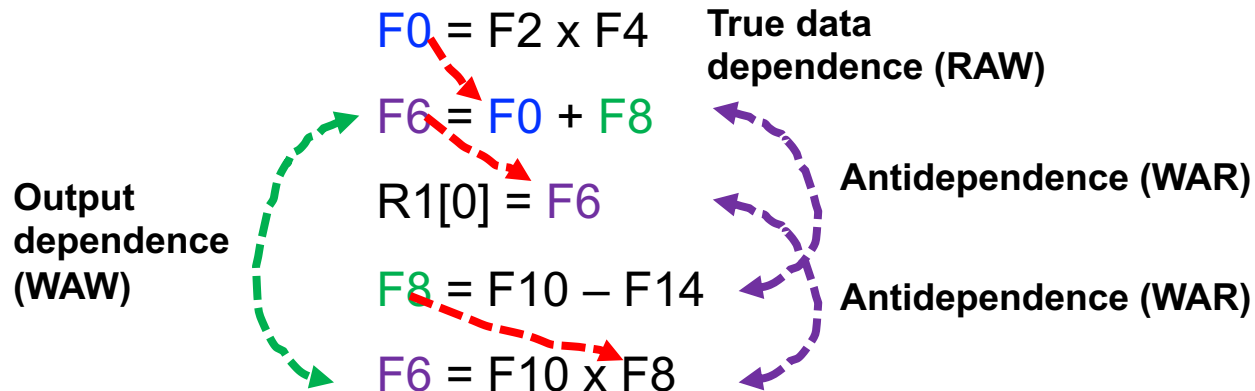
- Dynamic scheduling breaks the “in order” execution
  - **Out-of-order** execution :
    - Execute an instruction as soon as its data operands are available

|       |            |                                  |
|-------|------------|----------------------------------|
| DIV.D | F0,F2,F4   |                                  |
| ADD.D | F10,F0,F8  |                                  |
| SUB.D | F12,F8,F14 | ← Execute this instruction first |



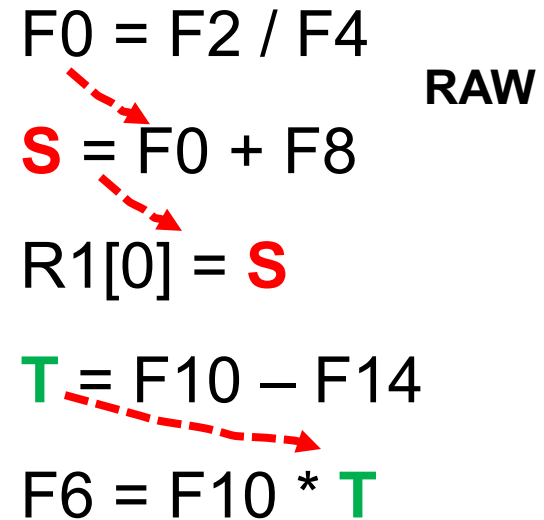
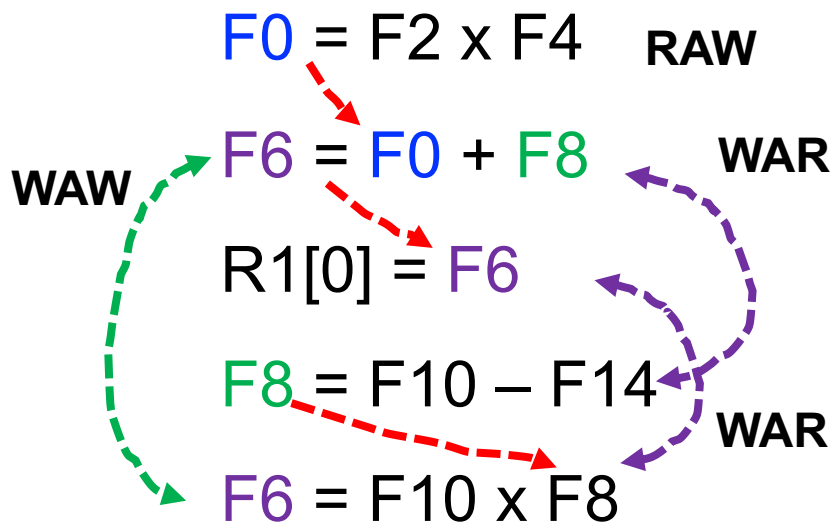
# New hazards

- **New hazards** to deal with
  - **WAR**
    - Possibility of overwriting a value that has not been read yet
  - **WAW**
    - Writing twice to the same location



# Register Renaming: Idea

- Rename the register in hardware to minimize the **WAW** and **WAR** hazards



Only RAW hazards remain

# Dynamic Scheduling Using Tomasulo's Algorithm

---

- Introduced by Robert Tomasulo
- Implemented in IBM 360/91 in its floating-point unit
  - IBM 360/91 had long memory and floating-point arithmetic delays
- Modern processors use a variation of Tomasulo's Algorithm



**IBM 360/91**



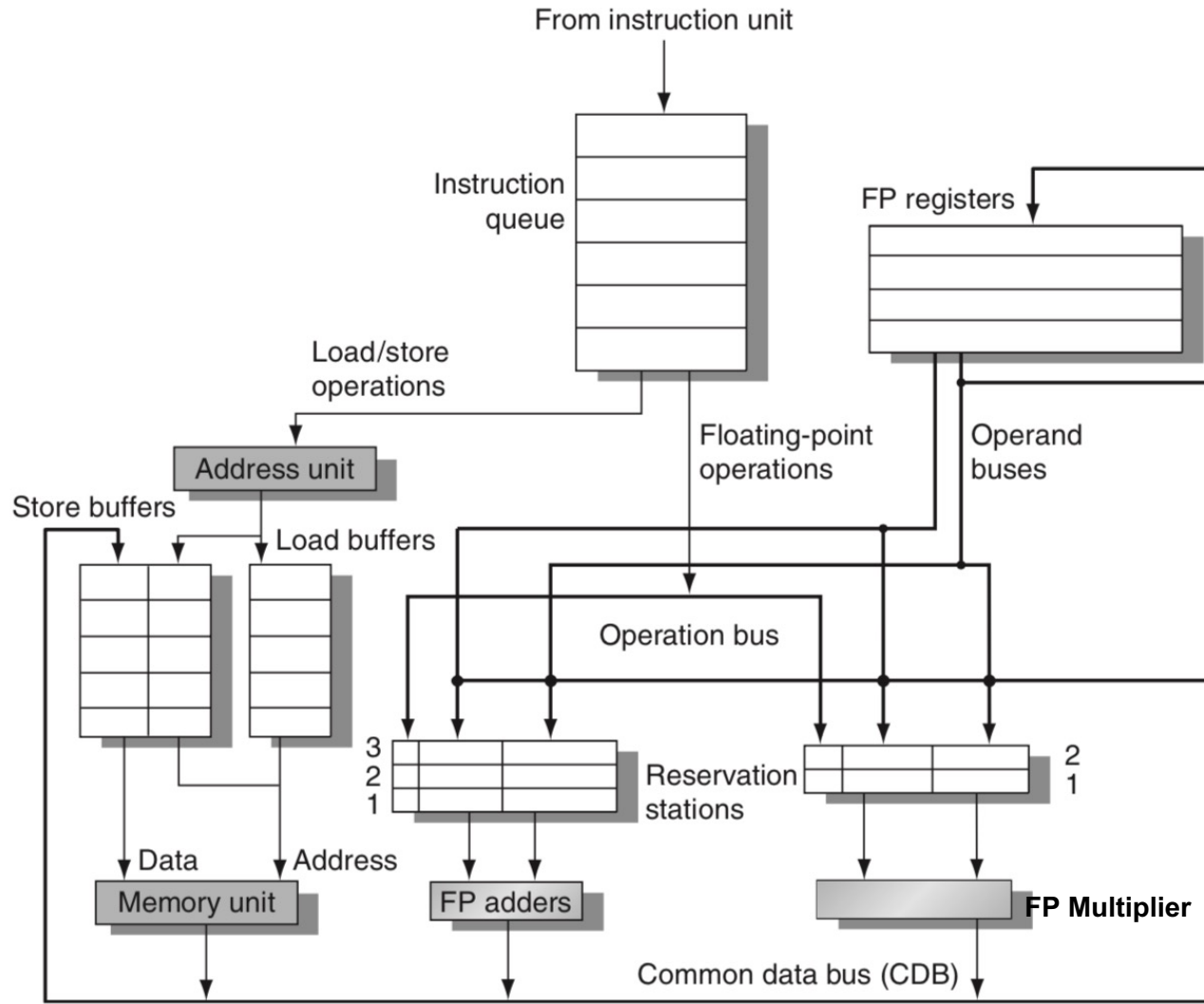
**Robert Tomasulo**

# Tomasulo's Algorithm

---

- Tomasulo's Algorithm allows...
  - Out-of-order execution
  - Tomasulo's algorithm can handle anti- and output dependences by **register renaming**
  
- In Tomasulo's Algorithm, each instruction goes through three steps
  - **Issue**
    - Instructions are stored into Instruction Queue
    - Transfer instruction to **RS (Reservation station)** if available
    - **Rename registers** to eliminate WAR and WAW
  
  - **Execute**
    - Monitor bus for new data and distribute it to instructions waiting for the data
    - Execute instructions in functional units when operands available
  
  - **Write result**

# Tomasulo's Algorithm (cont'd)



The basic structure using Tomasulo's algorithm

---

# **Limitations of ILP**

# The Limitations of ILP

---

- **Applications (algorithms)**

- Different applications (algorithms) have different numbers of instructions that can run simultaneously

- **Compiler sophistication**

- Need good compilers that can generate and/or schedule instructions that run in parallel

- **Hardware sophistication**

- Need complex hardware that can find more instructions to run in parallel

- **Maximum ILP is fundamentally limited by the RAW dependencies**

- Cannot issue more instructions if previous computations are not finished

# Upper limit of ILP with Perfect CPU

---

