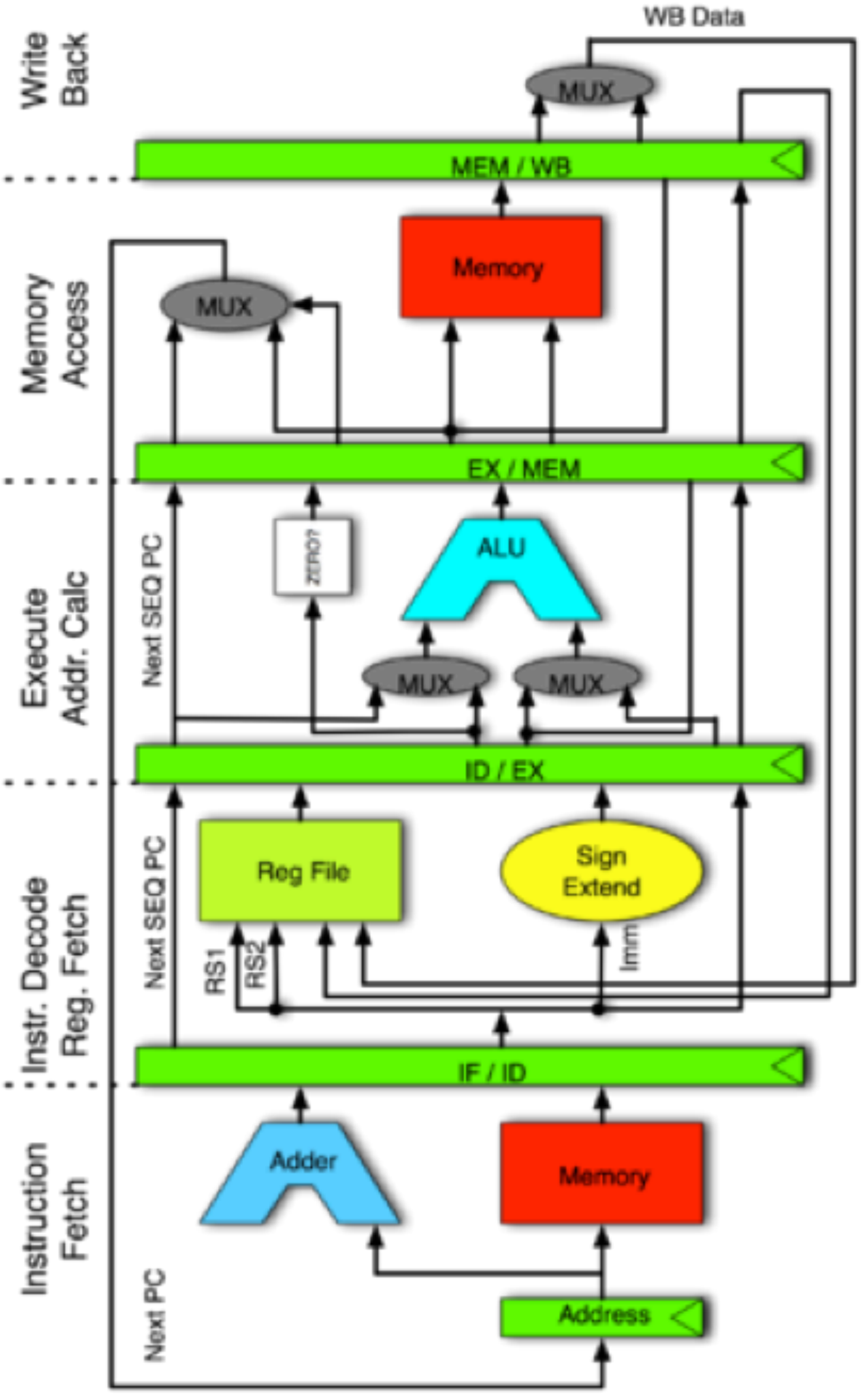
CS 520
Part II
Instruction Pipelines

Timothy N. Miller

Terminology

- **Instruction issuing**: D/RF -> EX
- **Scalar pipeline**: Issue one ins/cycle
- **Functional unit**: Pipeline stage or logic unit that implements operations


WB Data
Write Back
MUX
MEM / WB
Memory
Memory Access
MUX
EX / MEM
ZERO?
ALU
Next SEQ PC
MUX
MUX
Execute Addr. Calc
ID / EX
Reg File
Sign Extend
Next SEQ PC
RS1
RS2
Imm
Instr. Decode Reg. Fetch
IF / ID
Adder
Memory
Instruction Fetch
Next PC
Address

Terminology

- **Sequential Execution Model**
  - Execute logically in non-pipelined order
  - Results must match strictly sequential execution
  - Architectural PC -> current instruction
  - Data dependencies respect program order
  - Processor state updated in program order
    - Architectural registers, memory locations
  - Every pipelined processor implements this model

Terminology

- **ILP** (Instruction Level Parallelism): Parallelism available in instruction sequence
- **Machine parallelism**: Mechanisms in instruction pipeline that exploits ILP
  - Sufficient machine parallelism is required in order to approach CPI of 1.

Dependencies and Parallelism

- ADD R1,R2,R3
- ADD R4,R5,R6

- ADD R1,R2,R3
- ADD R4,R5,R1

APEX features recap

- Destination and source register addresses appear in fixed fields within instruction.
  - Logic in D/RF blindly assumes those fields are valid, *before* decode
  - If address is invalid, read data is discarded
  - Allows overlap of decode and register read

opcode (6)	rs (5)	rt (5)	rd (5)	sa (5)	function (6)

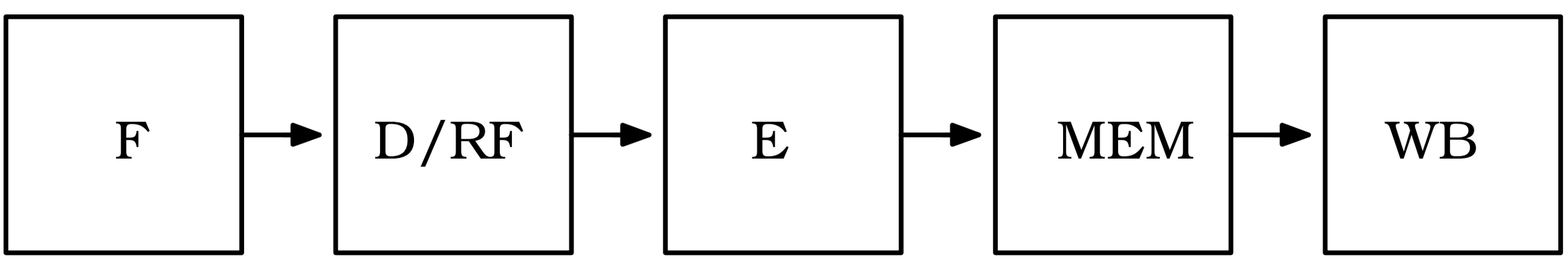
opcode (6)	rs (5)	rt (5)	immediate (16)

APEX features recap

- For some instructions, no processing is done in some stages
  - register-to-register and MEM stage
  - Store instruction and WB stage
- Registers always read in the same stage
  - Minimizes read ports

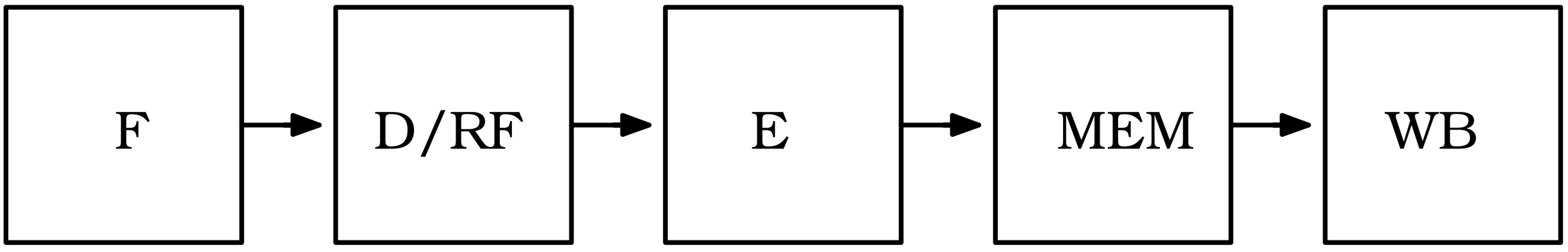
APEX features recap

- Register file written from same stage
  - Even if result is available early
  - Minimizes write ports



APEX features recap

- EX stage implements ALU and effective address for LOAD/STORE
- Ideally, processing delay of each stage is the same, T = clock period.
- Stalls occur when instruction spends more than one cycle in a stage



Reasons for stalls

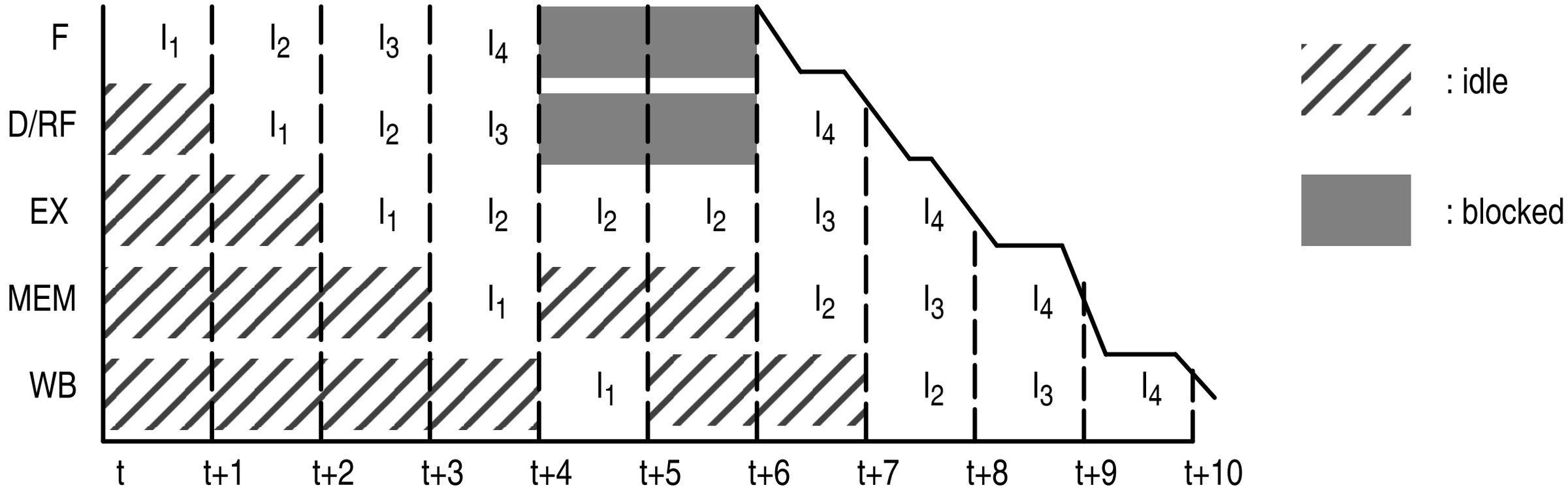
- Instruction fetch delay (I-Cache miss)
- Data access delay (D-Cache miss)
- Data dependencies

- Simple APEX pipeline: All stages stall together

Enhancing Parallelism: Multiple Functional Units

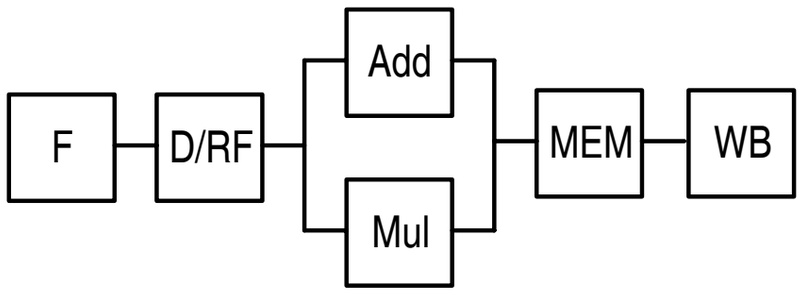
- A single non-pipelined functional unit can be a bottleneck if execution latencies vary between types of instructions.
- Example: MULtiply instruction with latency of 3 cycles, while all other ops have latency of 1.

```
I1: ADD R1, R2, R3 /* R1 <- R2 + R3 */
I2: MUL R8, R4, R6 /* R8 <- R4 * R6 */
I3: ADD R7, R2, R5
I4: ADD R9, R3, #1 /* R9 <- R3 + 1 */
```



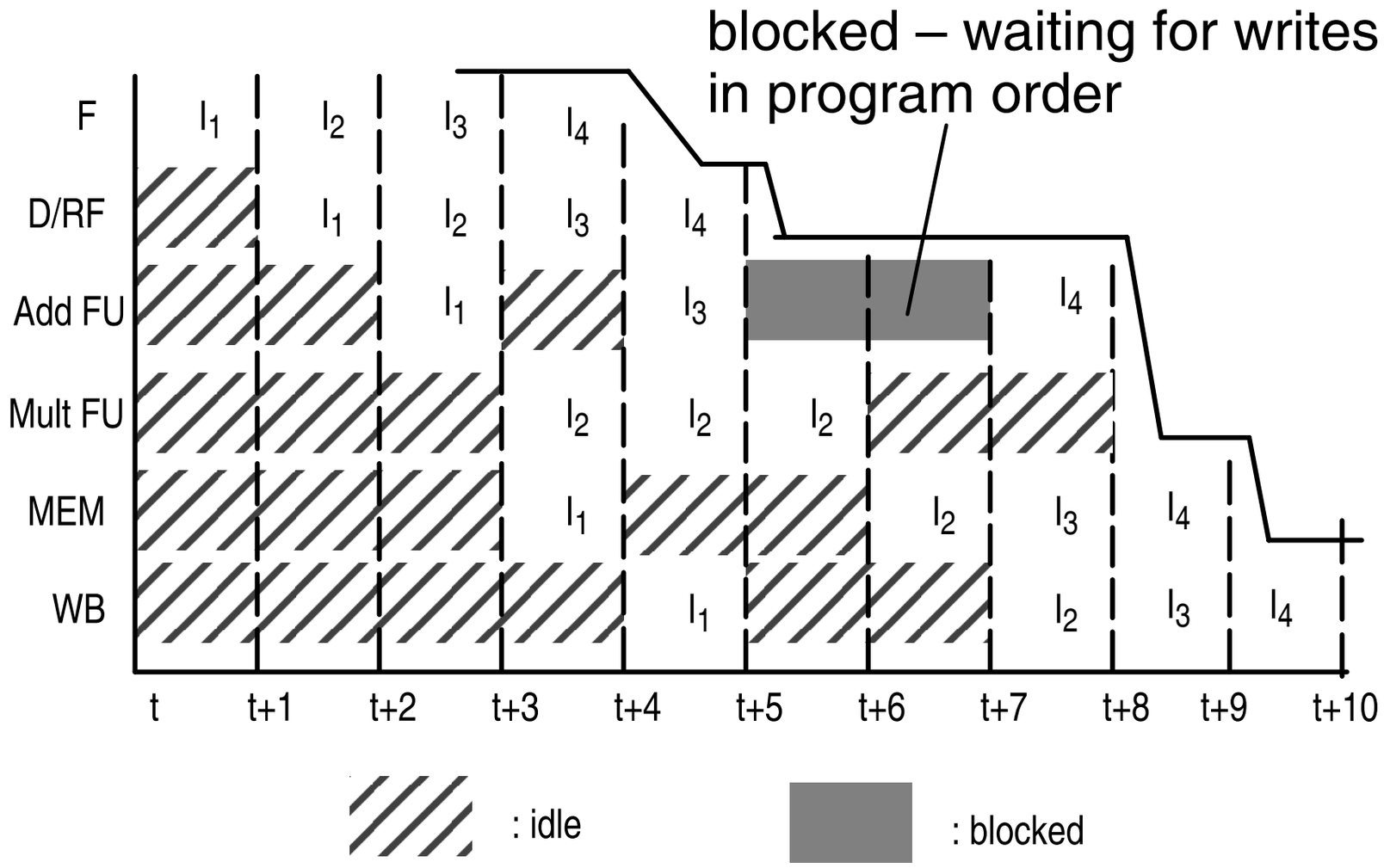
Multiple Functional Units

```
I1: ADD R1, R2, R3 /* R1 <- R2 + R3 */
I2: MUL R8, R4, R6 /* R8 <- R4 * R6 */
I3: ADD R7, R2, R5
I4: ADD R9, R3, #1 /* R9 <- R3 + 1 */
```

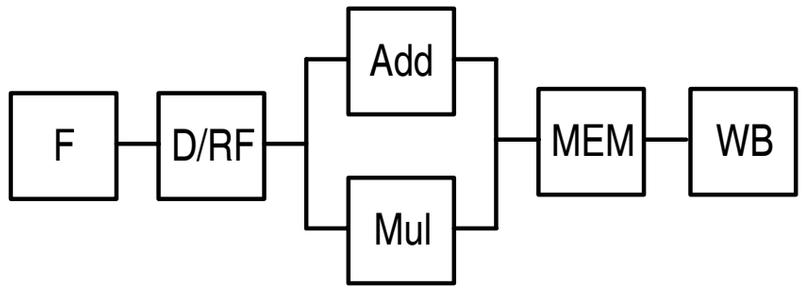


Add: FU for ADD instructions

Mult: FU for MUL instructions

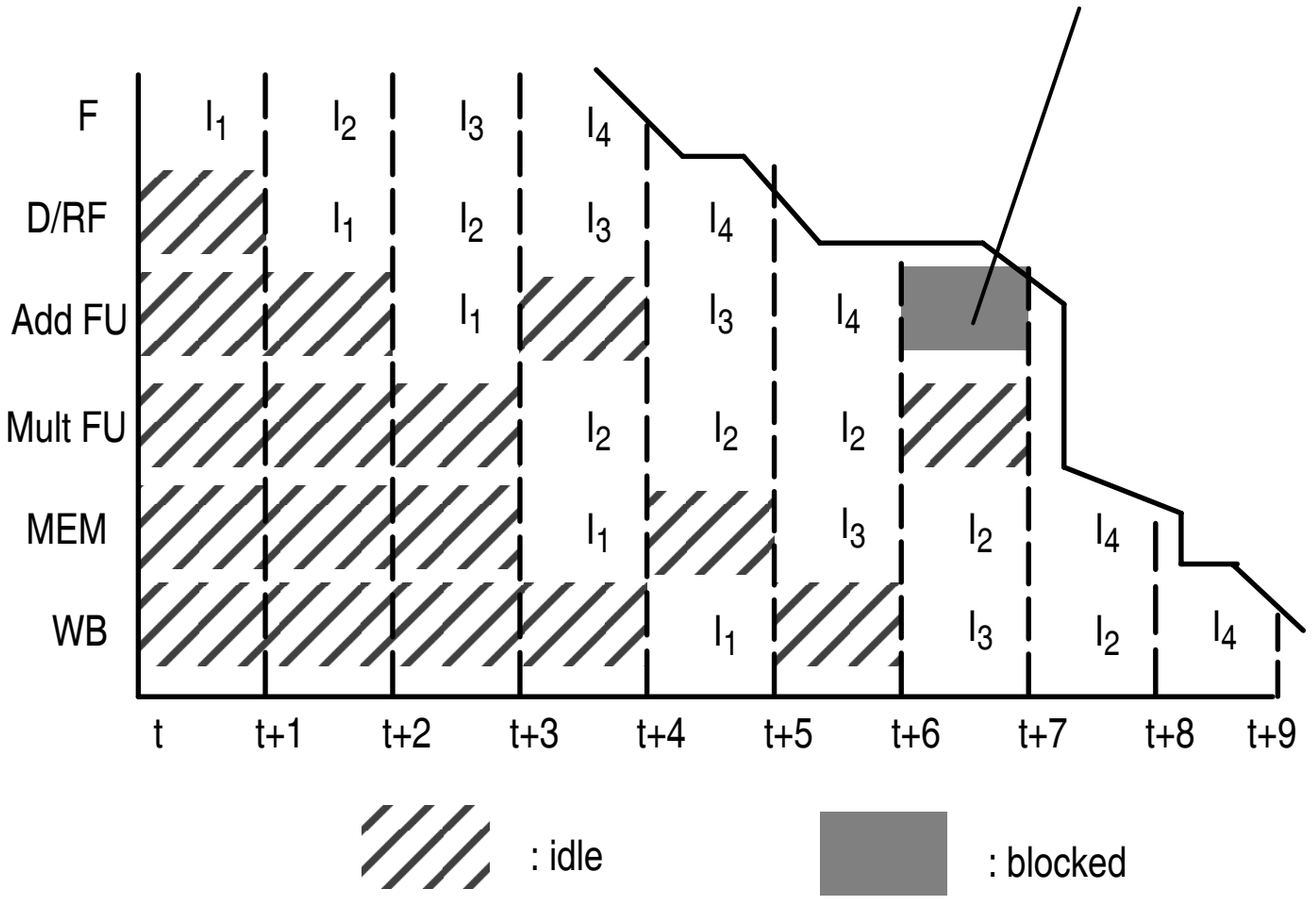


Out-of-order WB



Add: FU for ADD instructions

Mult: FU for MUL instructions



Mechanisms to support multiple FUs

- Additional data paths: D/RF -> FUs, FUs -> WB
- Mechanism to handle simultaneous and out-of-order completions
- Data forwarding from one FU to another

Contemporary pipeline Functional Units

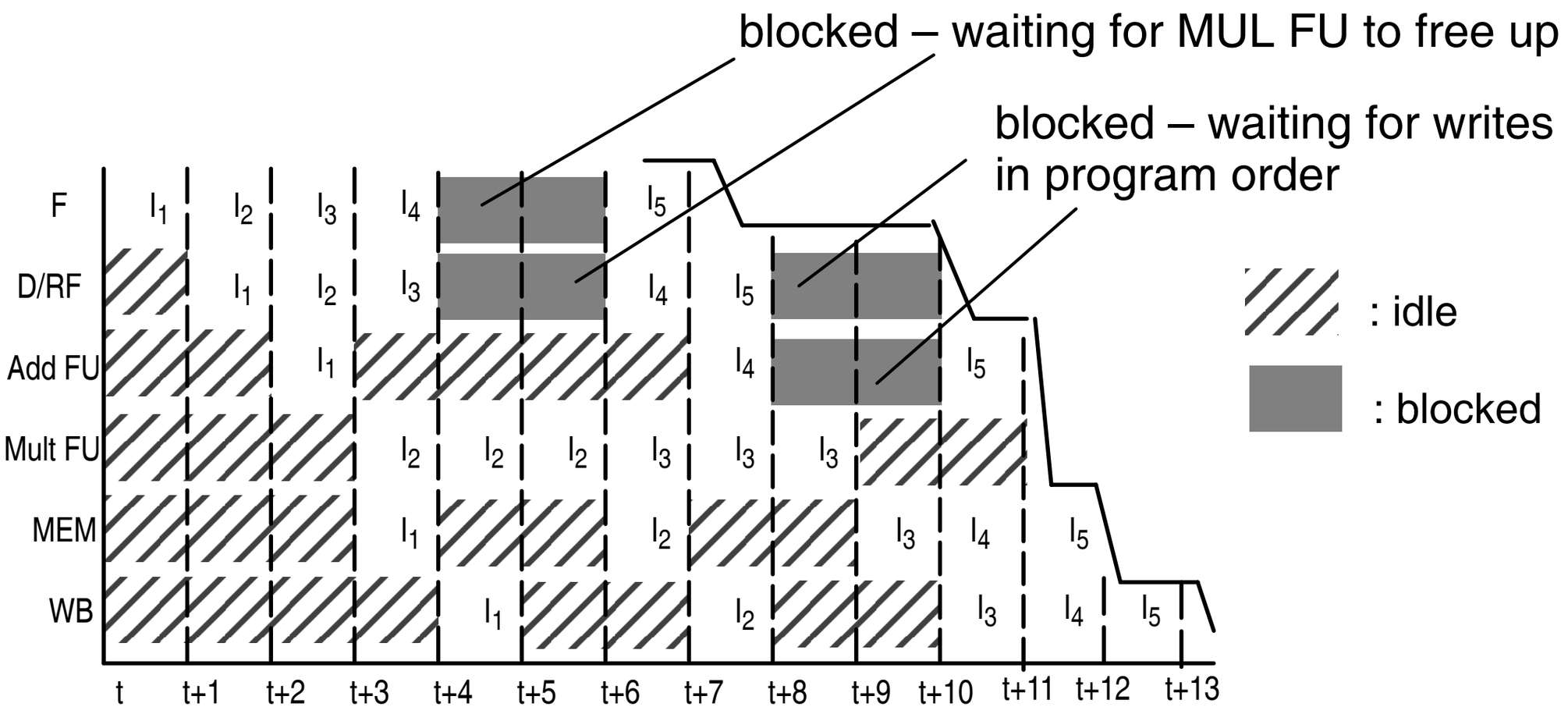
- Integer FU
  - Single-cycle latency for integer, logical, shift, and often also MUL
- Floating point FU
- Load/Store FU
  - Also implements address generation

Motivation for multiple FUs

- Avoid FU bottlenecks
- Break monolithic FU into multiple FUs, each implementing specific function -- allows hardware to be optimized
- Add/upgrade FUs in future designs
  - E.g. implement FP in software now, hardware later
- Design modularity
  - Selectively enhance only some FUs

Multi-cycle resource contention

I_1: ADD R1, R2, R3 /* R1 <– R2 + R3 */
I_2: MUL R4, R5, R6 /* R4 <– R5 * R6 */
I_3: MUL R7, R8, R9
I_4: ADD R10, R3, #10 /* R10 <– R3 + 10 */
I5: ADD R11, R12, R13



Removing Bottlenecks

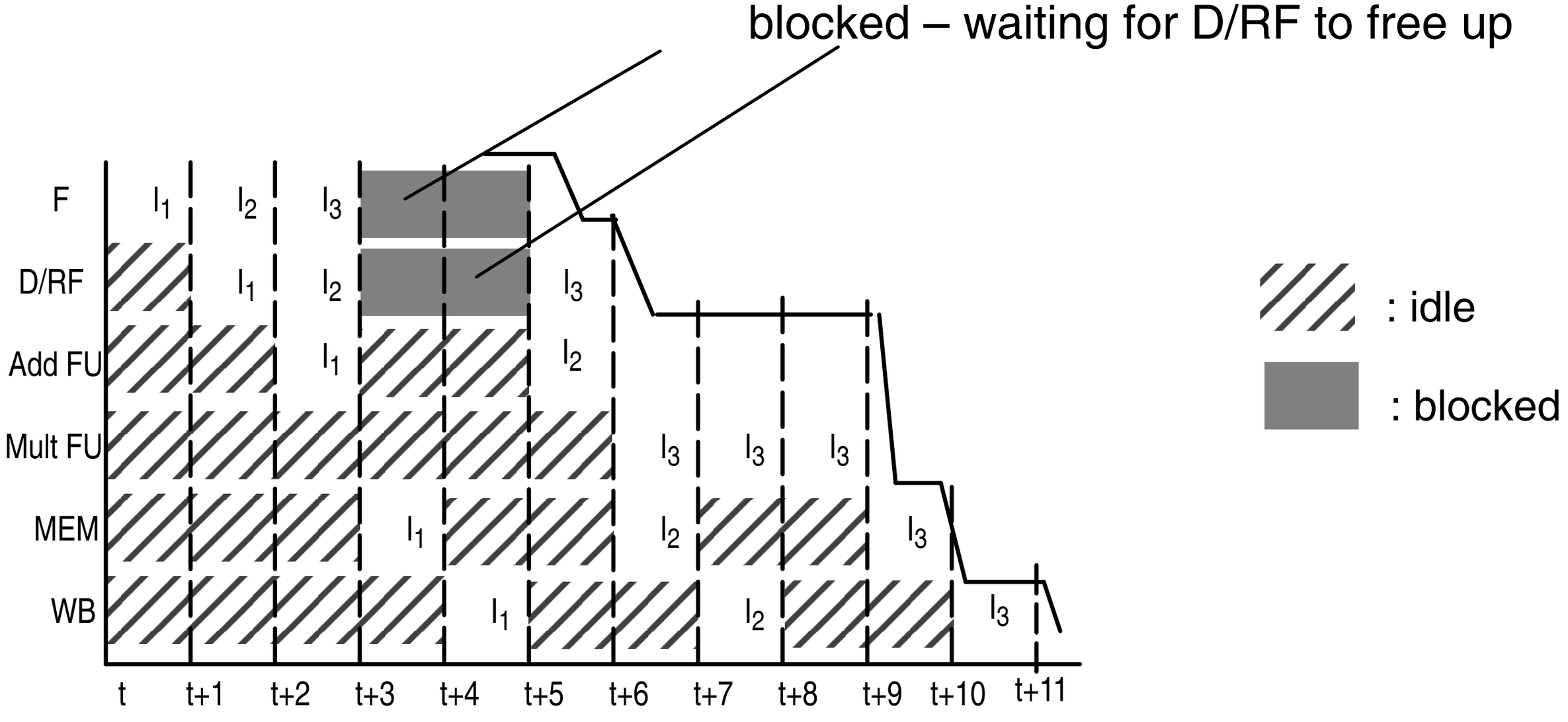
- MUL FU is the bottleneck
- Out-of-order reduces to 12 cycles
  - Indirect blocking: Need to write registers in program order
- Pipelining the MUL FU
  - Removes contention when multiples are not interdependent

I_1: ADD R1, R2, R3 /* R1 <– R2 + R3 */
I_2: MUL R4, R5, R6 /* R4 <– R5 * R6 */
I_3: MUL R7, R8, R9
I_4: ADD R10, R3, #10 /* R10 <– R3 + 10 */
I5: ADD R11, R12, R13

Issue Queues

- Conditions to issue so far:
  1. Required FU must be free
  2. All input operands available
- Let's relax #2: Issue without operands, execute when available

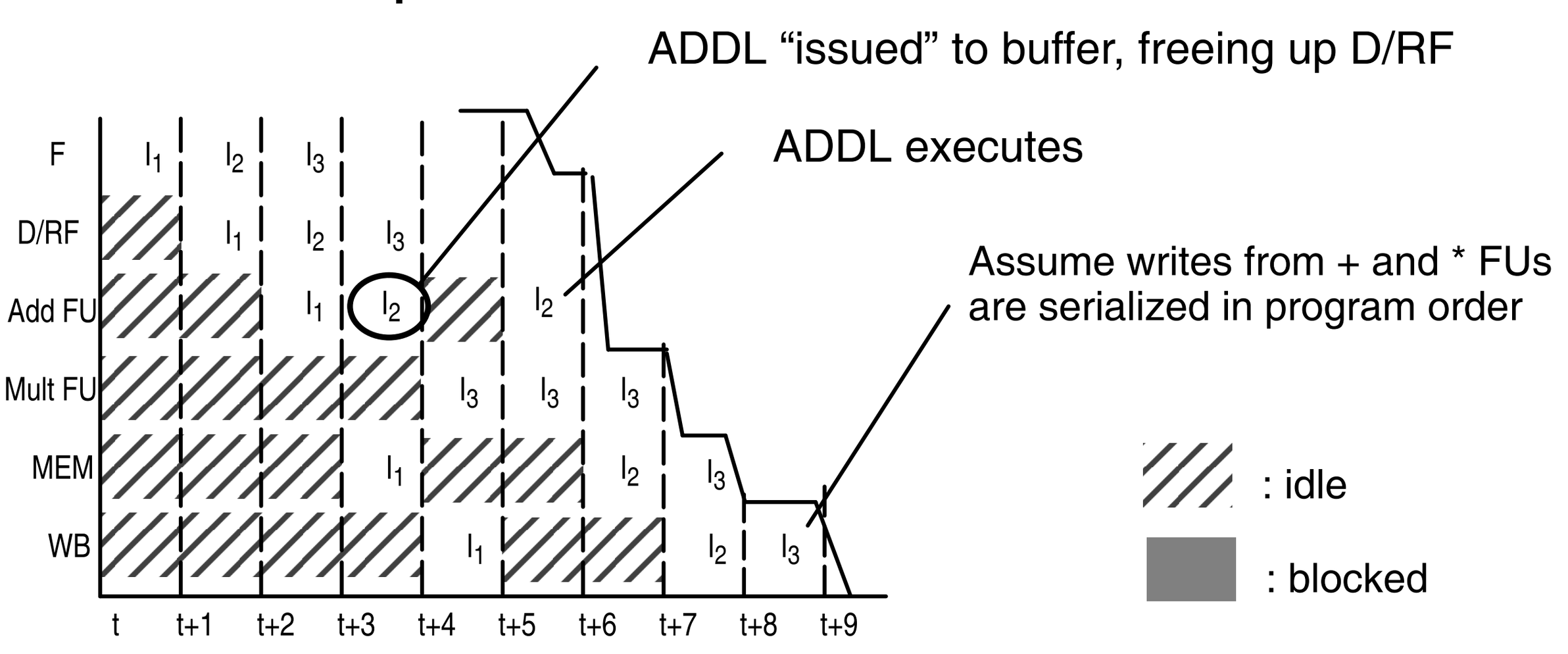
```
/* R2, R6, R7 contain valid data at this point */
I1      LOAD   R1, R2, #10
I2      ADDL   R4, R1, #1
I3      MUL    R5, R6, R7
```



Issue Queue

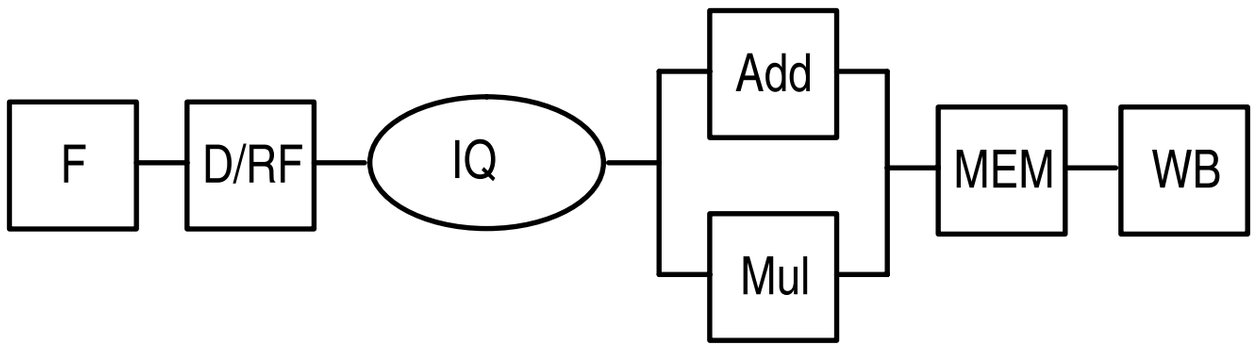
```
I1      LOAD   R1, R2, #10
I2      ADDL   R4, R1, #1
I3      MUL    R5, R6, R7
```

- Move ADDL out of D/RF, waiting for R1 in issue queue.



Issue Queue

- D/RF is a critical resource
- Free up D/RF to be used by later instructions
- Now execution can *start* out of order.
- Other names: scheduling queue, instruction window buffer, reservation station



Add: FU for ADD instructions

Mult: FU for MUL instructions

Mechanisms required for Issue Queue

- Ensure storage is available in IQ
- Note sources not available and hold value of already-valid operands
- Receive inputs as they become available
- Start execution when all inputs received

- Decouples F and D/RF stages from rest of pipeline.
  - Front-end -> IQ -> back-end

Modifications to D/RF

- Decode instruction
- Read registers already available
- Note dependencies on earlier instructions in program order, particularly for unavailable inputs.
- Configure data paths that will resolve dependencies
- Move Instruction to IQ

Dispatch & Issue

- **Dispatch** -- Above steps to add instruction to issue queue
- **Issue** -- Starting up execution when inputs are received and FU is available

- Opens up **Instruction Window**
  - Selectively issue out-of-order
  - Maximize exploitation of ILP

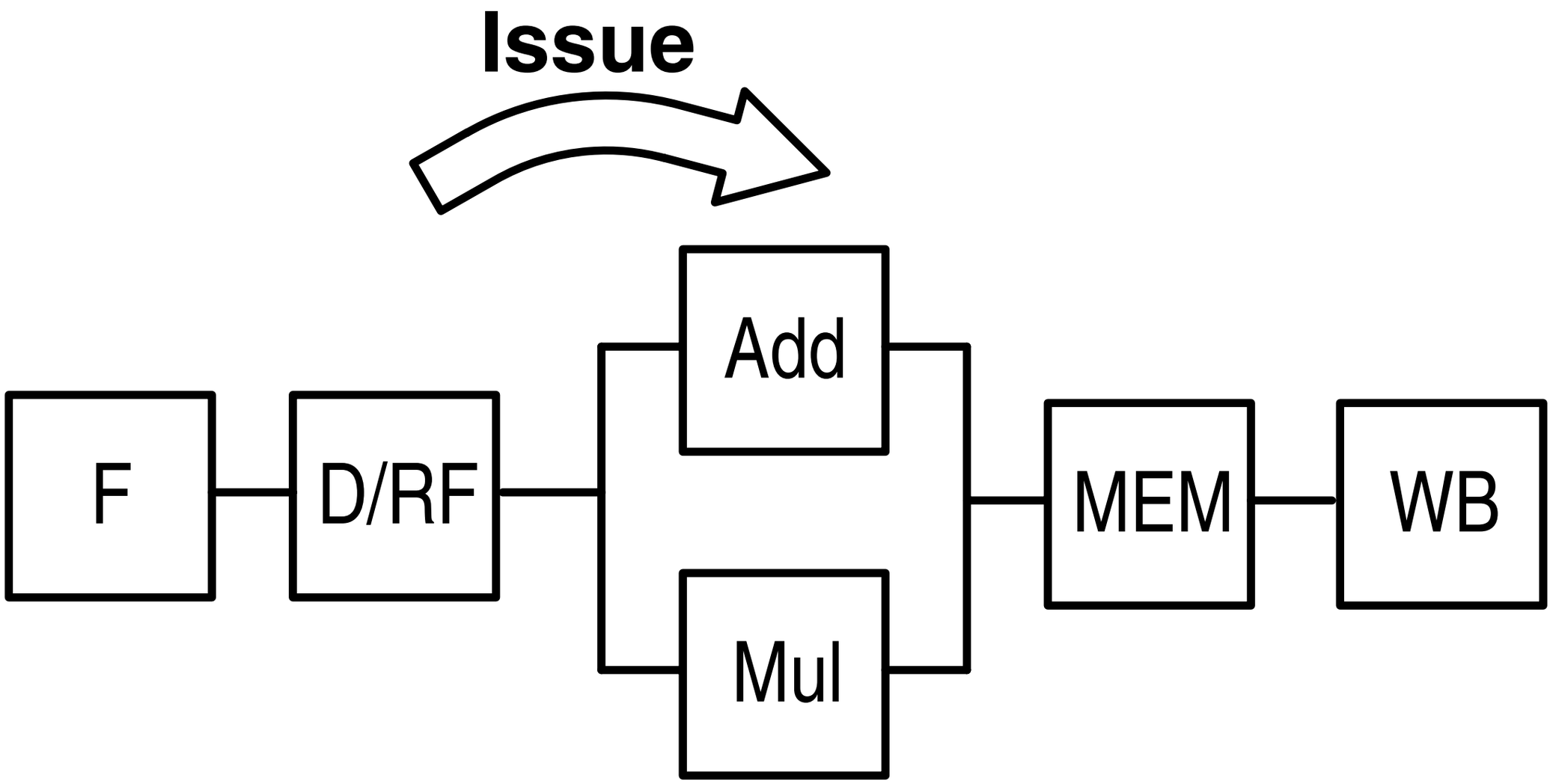
Requirements to Dispatch — Before

- Conditions to issue:
  1. Required FU must be free
  2. All input operands available
- Let's relax #2: Issue without operands, execute when available

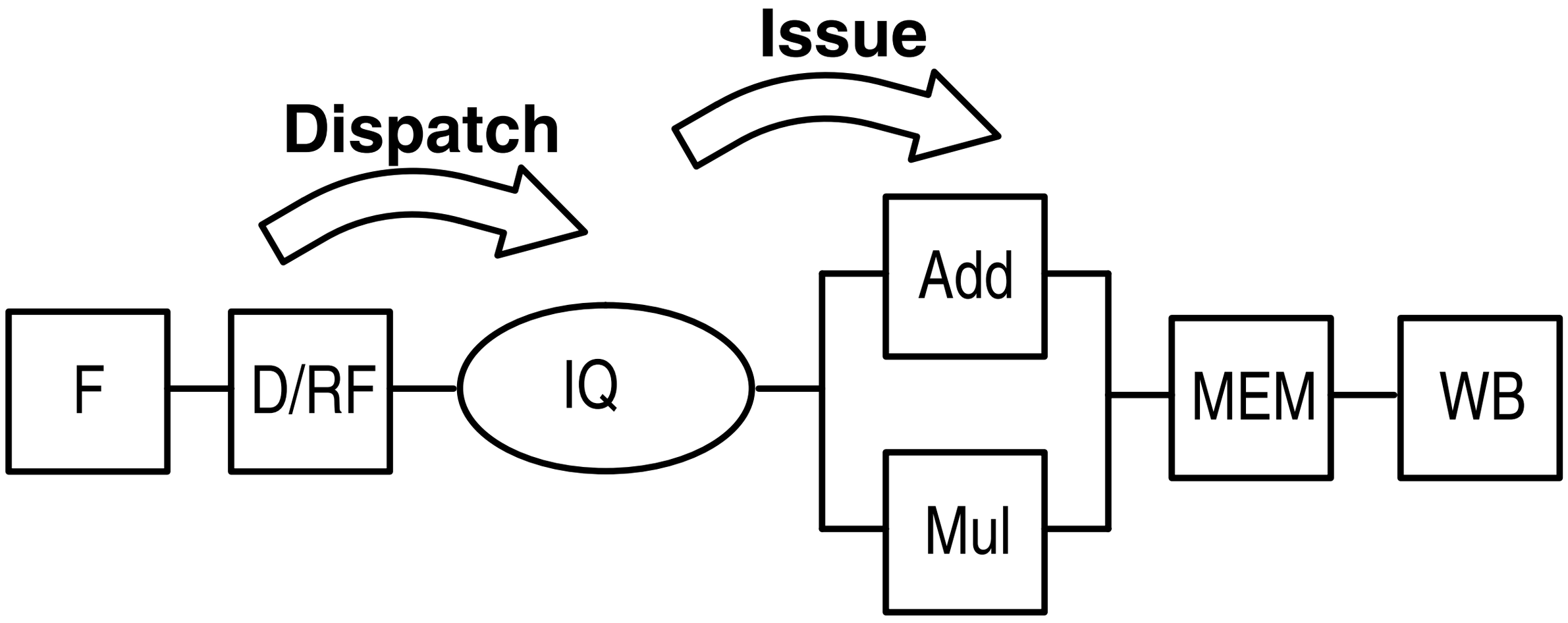
Requirement to Dispatch — Now

- Only requirement: Empty slot in IQ
- Conditions now relaxed:
  1. Required FU must be free
  2. All input operands available

In-order issue



Out-of-order Issue



Centralized Implementation of IQ

- Global IQ (shelves)
- Waiting instructions carry tag indicating required FU
- Multiple connections from IQ to FUs
  - Avoids serializing startups

Distributed Implementation of IQ

- Separate instruction buffers (reservation stations) for each FU
- Each reservation station entry (RSE) for each FU defines a virtual FU (VFU)
  - Dispatch is issue to a VFU

Recap

- Multiple FUs of different types
- Pipelining of FUs
- Out-of-order completion (WB)
- IQ and dispatch/issue -- out-of-order startup

Terminology

- ***Resolving dependencies***: Setting up data flow paths to satisfy data-flow dependencies.
- ***Satisfying dependencies***: Completion of data-flow.
- ***Instruction dispatching***: Decoding instructions, determining VFUs needed, identifying dependencies.
- ***Instruction issuing***: Satisfying dependencies, setting up execution as physical FUs are available.
- ***Out-of-order processor***: Processor that supports OOO startup and completion to exploit ILP.

Wait States in OOO Machines

- WD - Waiting to be dispatched, when IQ slot (VFU) is not available.
- WOP - Waiting on one or more operand in IQ.
- WE - Waiting issue; all inputs available, waiting on free FU.
- WR - In execution, waiting on results (min 1 cycle)
- WWB - Execution completed, waiting on write-back
- WC - Waiting for commitment; prior writes must be committed in order to ARF and memory for sequential execution model.

Data Dependencies

- Given a pipeline with infinite parallelism, **program data dependencies** limit performance over non-pipelined execution.
- **Sources** (inputs) -- registers & memory needed by instructions
- **Input set** (read set) -- set of an instruction's inputs.
- **Sinks** (destinations, outputs) -- registers & memory updated by instructions
- **Output set** (write set) -- set of outputs.
- Some inputs and outputs can be implicit.

Dependency Examples

(a)

```
ADD R1, R4, R6    /* R1 <-- R4 + R6 */
```

Sources: R4, R6; read set = {r4, R6}
Destinations: R1, PSW flags (implicit);
Write set = {R1, PSW flags}

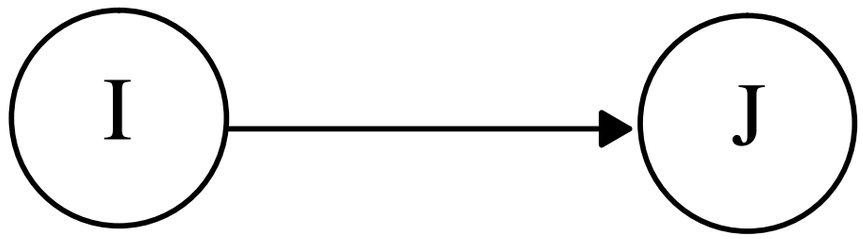
(b)

```
LOAD R1, R2, #120 /* R1 <-- Mem[R2 + 120] */
```

Sources: R2, Mem[R2 + 120];
Read set = {R2, Mem[R2 + 120]}
Destinations: R1;
Write set = {R1}

Sequential Ordering (Program Order)

- I > J -- I processed before J
- *True dependency, flow dependency, R-A-W*:
  - I precedes J in program order (I > J)
  - A source for J ≡ Dest for I



Example:

I1: ADD R1, R2, R4

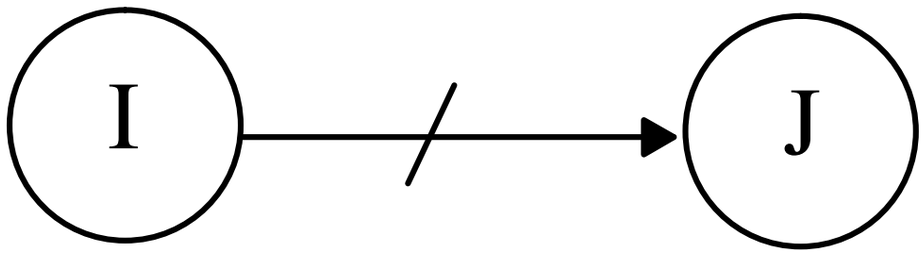
I2: MUL R4, R1, R8

Data Dependencies

- In-order startup, I > J:
  - Potential waiting of J in D/RF for result from I.
- Out-of-order startup, I > J:
  - Potential waiting of J in WOP state when I is in WE or E state (or even WWB)

Anti-dependency

- W-A-R dependency
  - I > J
  - Dest of J ≡ Source for I



Example:

I1: MUL R1, R6, R9
I2: ADD R6, R11, R15
I3: STORE R1, R10, #10

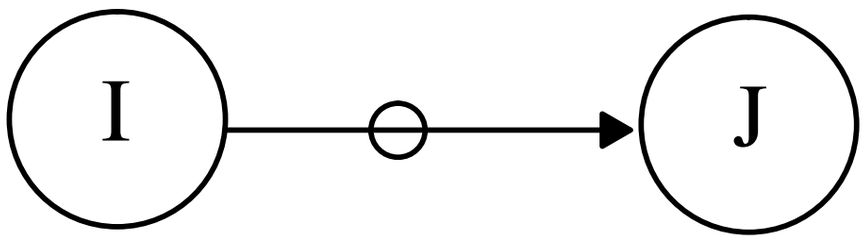
Anti-dependency

WD
WOP
WE
WR (E)
WWB
WC

- Anti-dep from I to J over Rk:
  - I must read Rk before J writes
  - I can force J to wait in WWB while I is in WOP.
  - (If I > J, and J is in WWB, I cannot be in any state that precedes WOP.)

Output dependency

- W-A-W dependency
  - I > J
  - Dest of I ≡ Dest of J



Example:

I1: MUL R1, R2, R6
I2: STORE R1, R5, #4
I3: ADD R1, R10, R19

Output dependency

WD
WOP
WE
WR (E)
WWB
WC

- Results must be written in program order
- I > J -- J can wait in WWB while I is in WOP, WE, or E.
- Results produced OOO must be held in temporary buffers.

Flow dependencies are fundamental

- Maintain anti- and output dependencies ONLY to respect flow dependencies.
  - Can ignore these using temporary storage mechanisms.
  - Any mechanism that respects **true data dependencies** is sufficient.
- Common cause of false dependencies:
  - Register reuse

Register dependencies

- Easy to detect
  - Statically, at compile time
  - Dynamically, in hardware
- Avoidance: Large numbers of registers

Memory dependencies

```
I1:    LOAD R1, R2, #10
I2:    ADD R4, R3, R1
I3:    STORE R4, R2, #10
```

- Load Mem[R2 + 10]
- Store Mem[R2 + 10]
- Anti-dependency

Memory dependencies

```
I1:    STORE R0, R5, #0
I2:    ADD R3, R1, R4
I3:    LOAD R1, R2, #10
```

- [R5+0] ?=? [R2+10]
- Hard or impossible to detect statically
- Heuristic detection: R2 and R5 point to different arrays?
- General techniques: Memory disambiguation

Memory dependencies

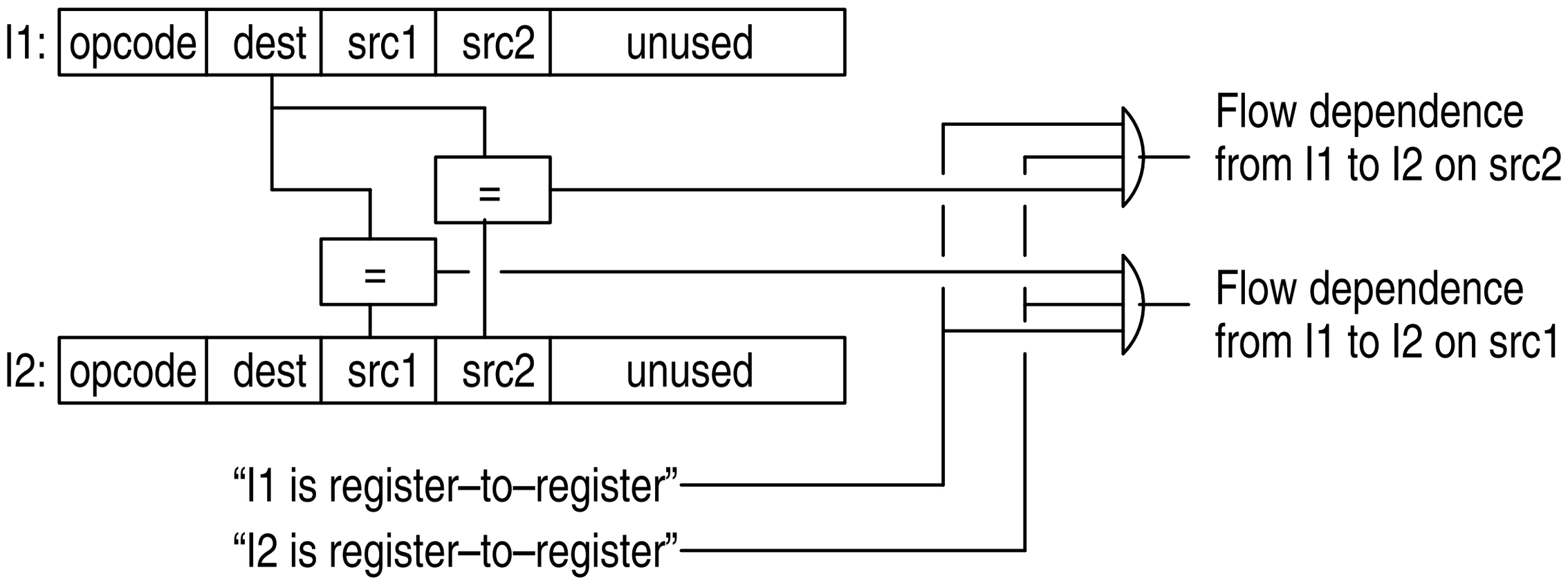
- Conservative: Process all memory accesses in order
- Dynamic: Wait until EAs have been computed, then reorder
- Speculative: Hoist reads before writes, squash on collision

Detecting dependencies dynamically

- Registers
  - 5 to 6 bit comparator
  - No ambiguities
- Memory
  - 32 to 64 bit comparators
  - Limited LSQ sizes
  - Because CAMs are expensive

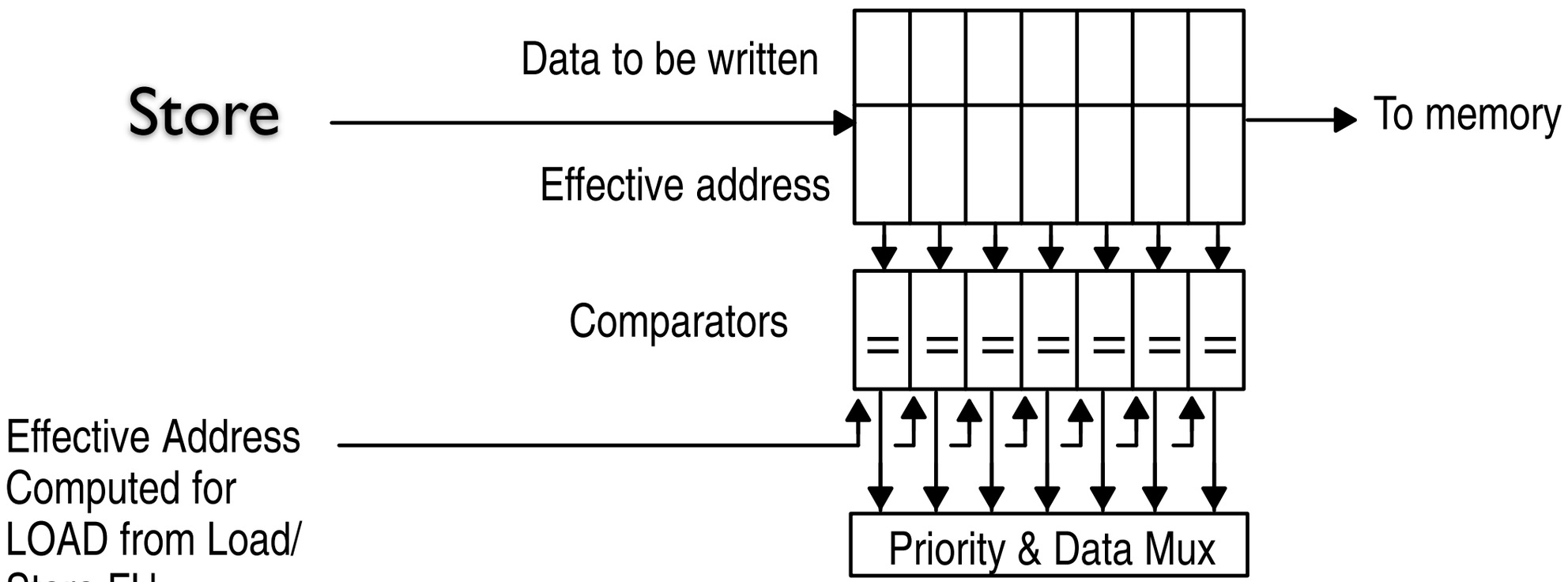
Register data dependencies

I1: ADD R1, R2, R4
I2: MUL R6, R8, R1



Memory data dependencies

- Writes to D-cache queued in **writeback buffer**, in FIFO order.
- Load addresses compared in parallel to all queued writes.



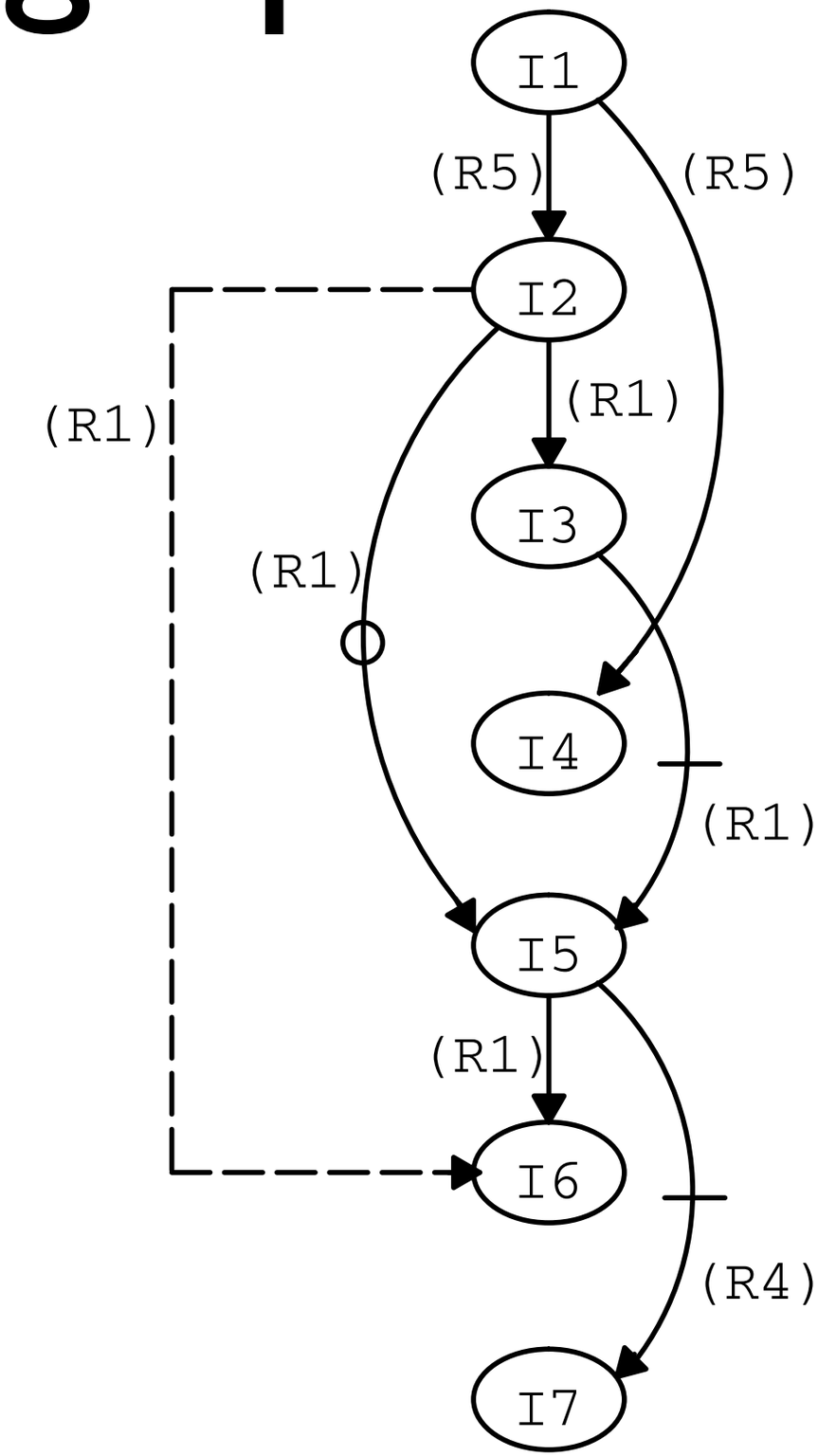
Recap

- Sequential execution model
- Stalls: Cache misses, data dependencies, multi-cycle functional units
- Breaking ALU into multiple specialized functional units
- ILP and Machine parallelism through pipelining and out-of-order execution.
  - Reducing bottlenecks
- Issue queues: Decouple front-end from back-end, reduce stalls in decode
  - Dispatch, Issue, instruction window
  - VFUs, RSEs
- Dependencies: True/Data, Anti-, Output

Data flow graphs

```
I1:     FMUL    R5, R7, R8
I2:     FADD    R1, R2, R5
I3:     MOV     R9, R1
I4:     FADD    R6, R5, R8
I5:     FDIV    R1, R4, R11
I6:     FST     R1, R10, #100
I7:     FADD    R4, R7, R11
```

Source



Coping With Data Dependencies

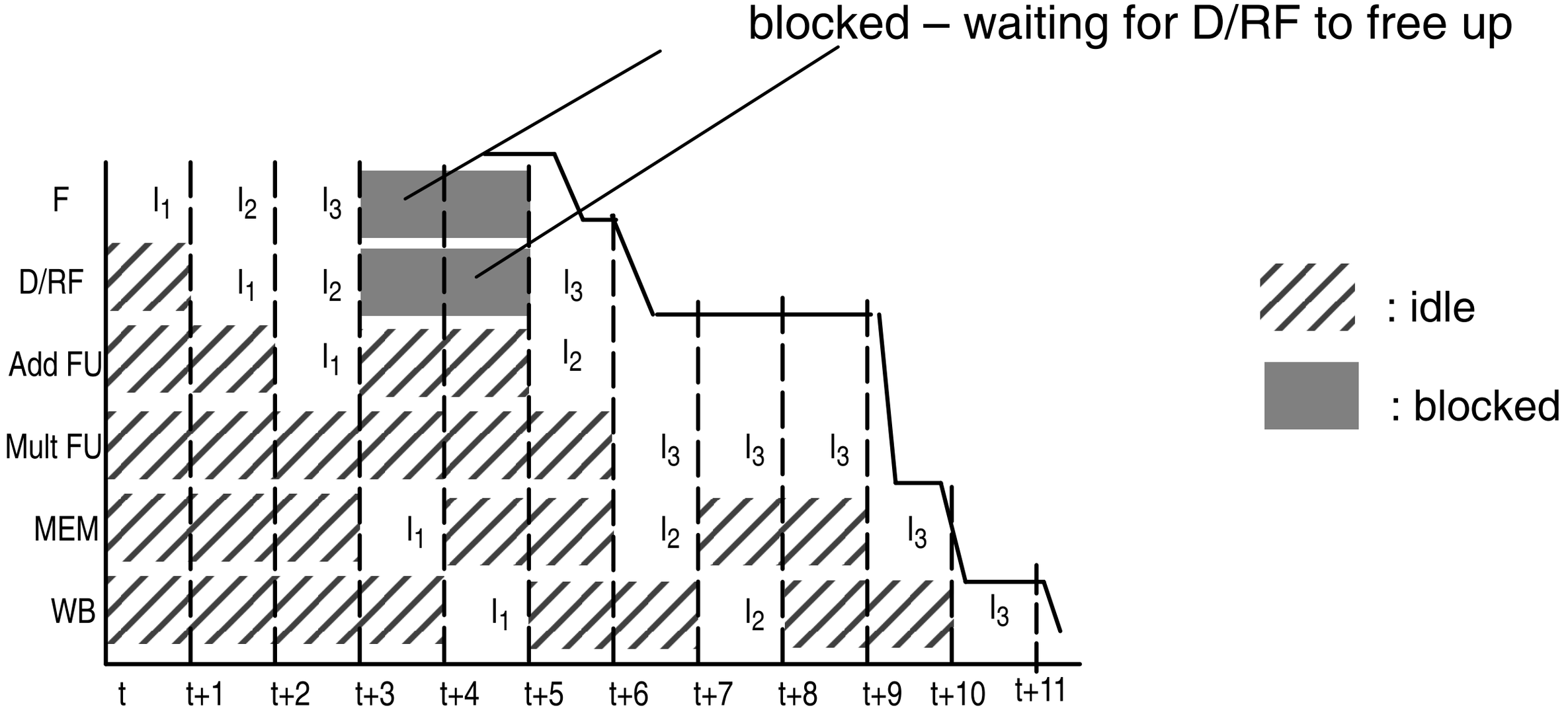
- Hardware techniques
  - Data forwarding
  - Dynamic instruction scheduling
  - Decoupled execute-access
- Software techniques
  - Software interlocking
  - Software pipelining

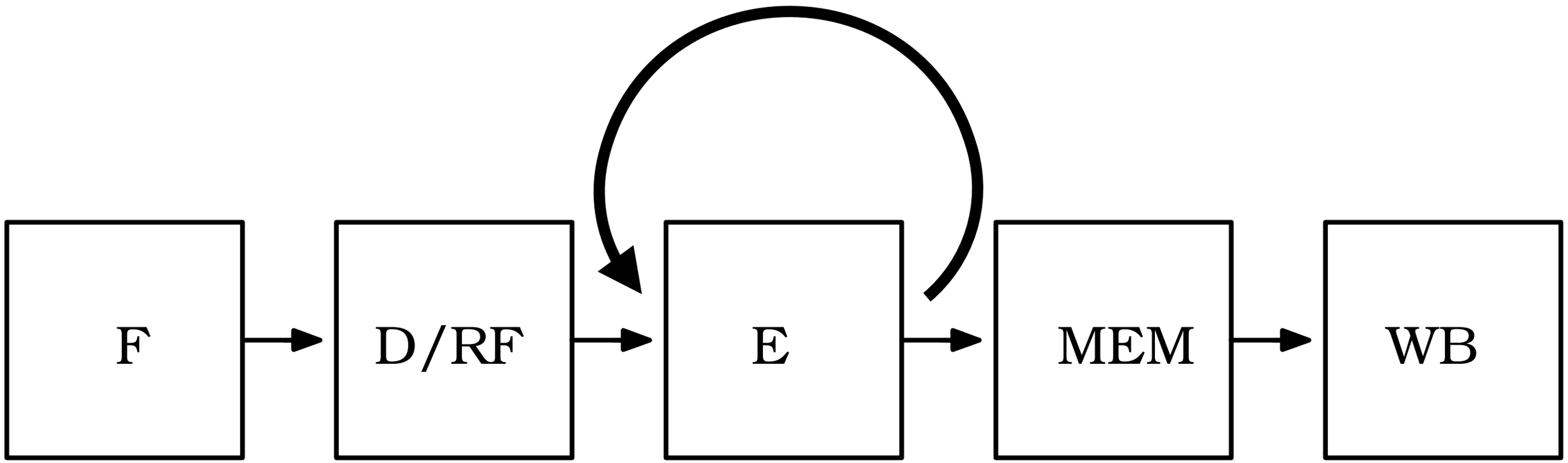
Data forwarding

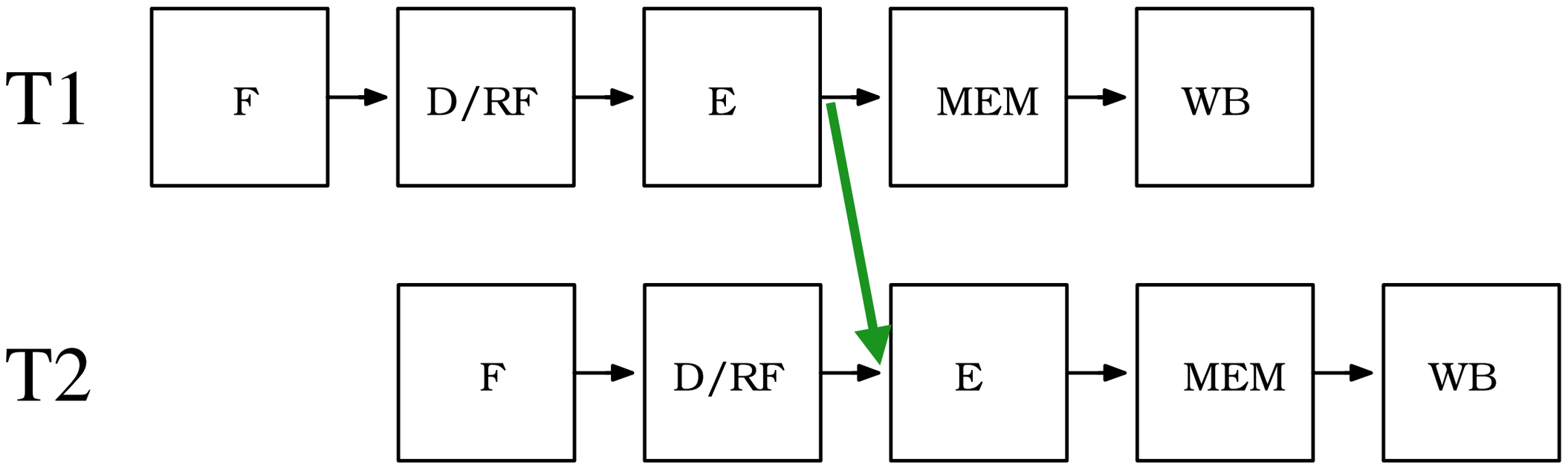
- (Shortstopping, data bypassing)
- Reduce delays due to flow dependencies
  - Pick the needed result out of the pipeline
    - Carry source reg addresses up to EX
    - Compare against dest reg into EX

```
I1:    AND R1, R2, R3
I2:    ADD R4, R1, R6
```

```
/* R2, R6, R7 contain valid data at this point */
I1      LOAD   R1, R2, #10
I2      ADDL   R4, R1, #1
I3      MUL    R5, R6, R7
```

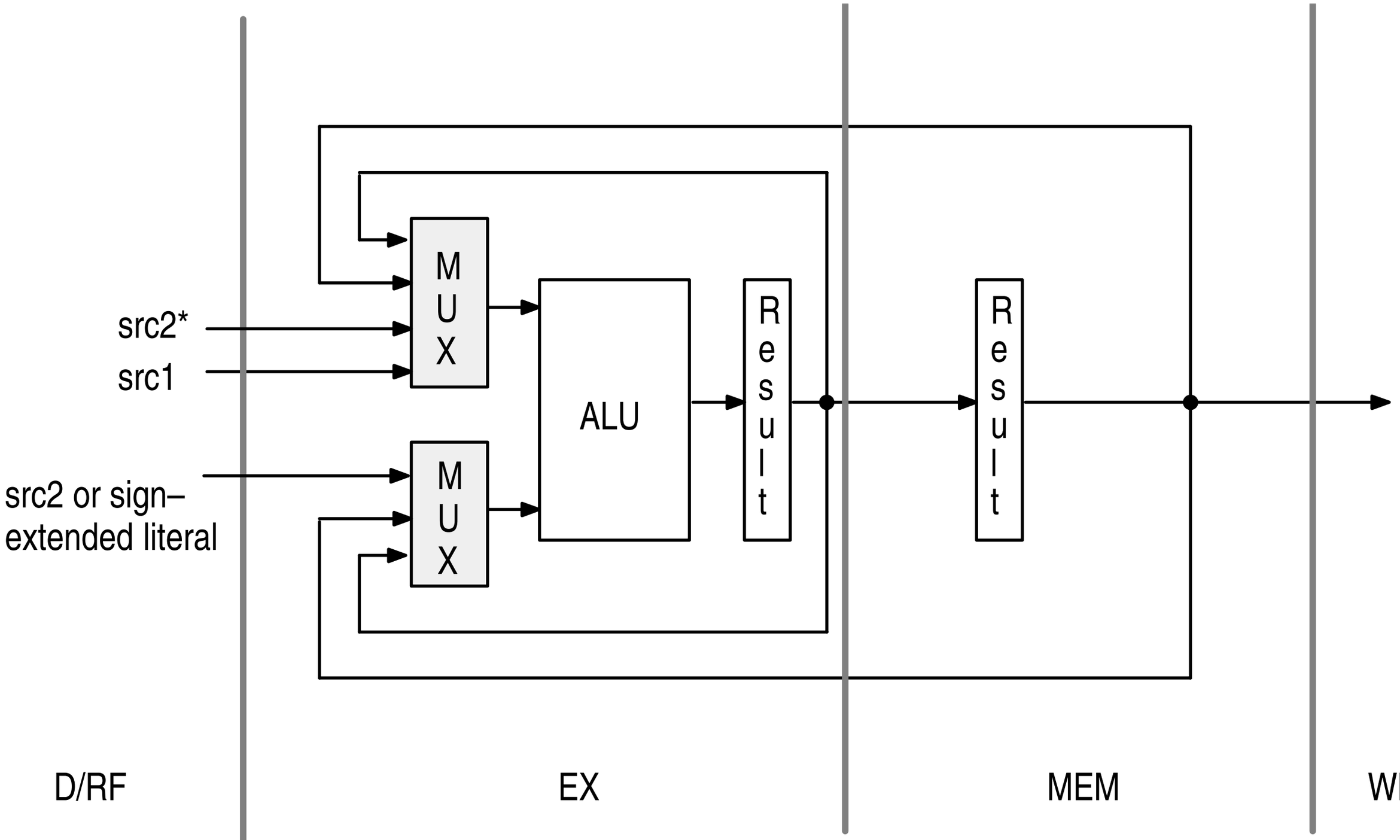



F
D/RF
E
MEM
WB


T1
F
D/RF
E
MEM
WB
T2
F
D/RF
E
MEM
WB

Forwarding paths

- Multiplexers and additional connections
- Logic to detect opportunities for forwarding
- In D/RF, choose between register and data from one of two potential forwarding sources (instructions in EX and MEM)
- Prioritize and ensure sources are valid



Paths Used for Forwarding th Result Computed by the ALU in APEX
(*applies to STORE instruction)

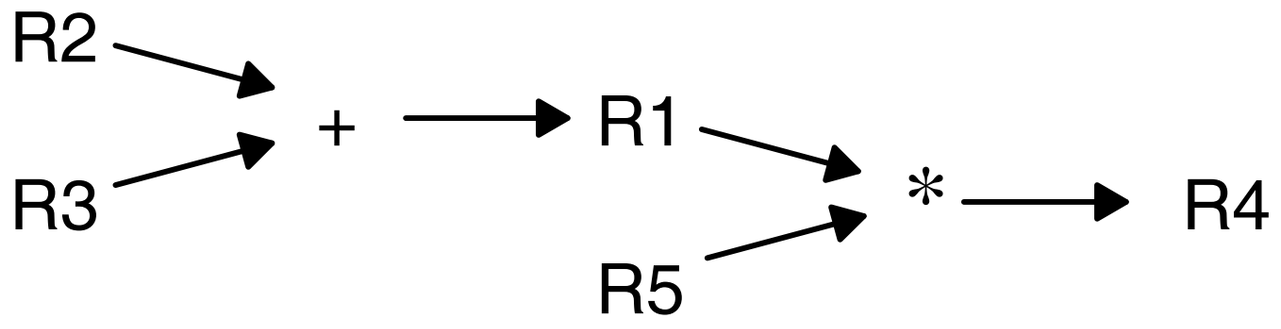
opcode (6)	rs (5)	rt (5)	rd (5)	sa (5)	function (6)

opcode (6)	rs (5)	rt (5)	immediate (16)

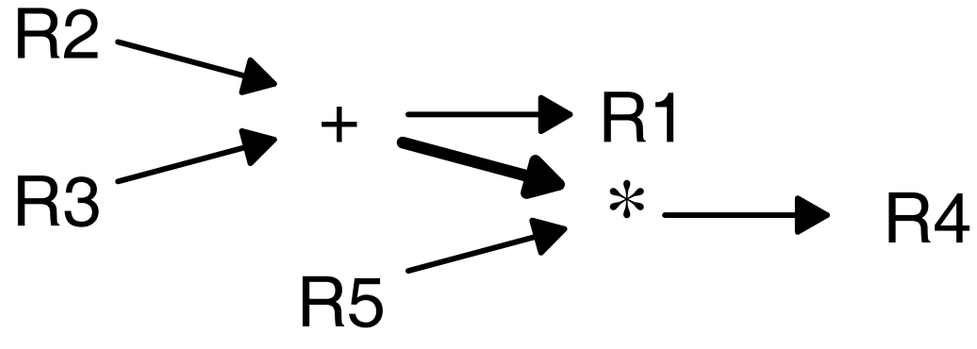
Direct dataflow between instructions

```
I1:   ADD R1, R2, R3
I2:   MUL R4, R5, R1
```

Original Data Flow:



Data Flow with Forwarding

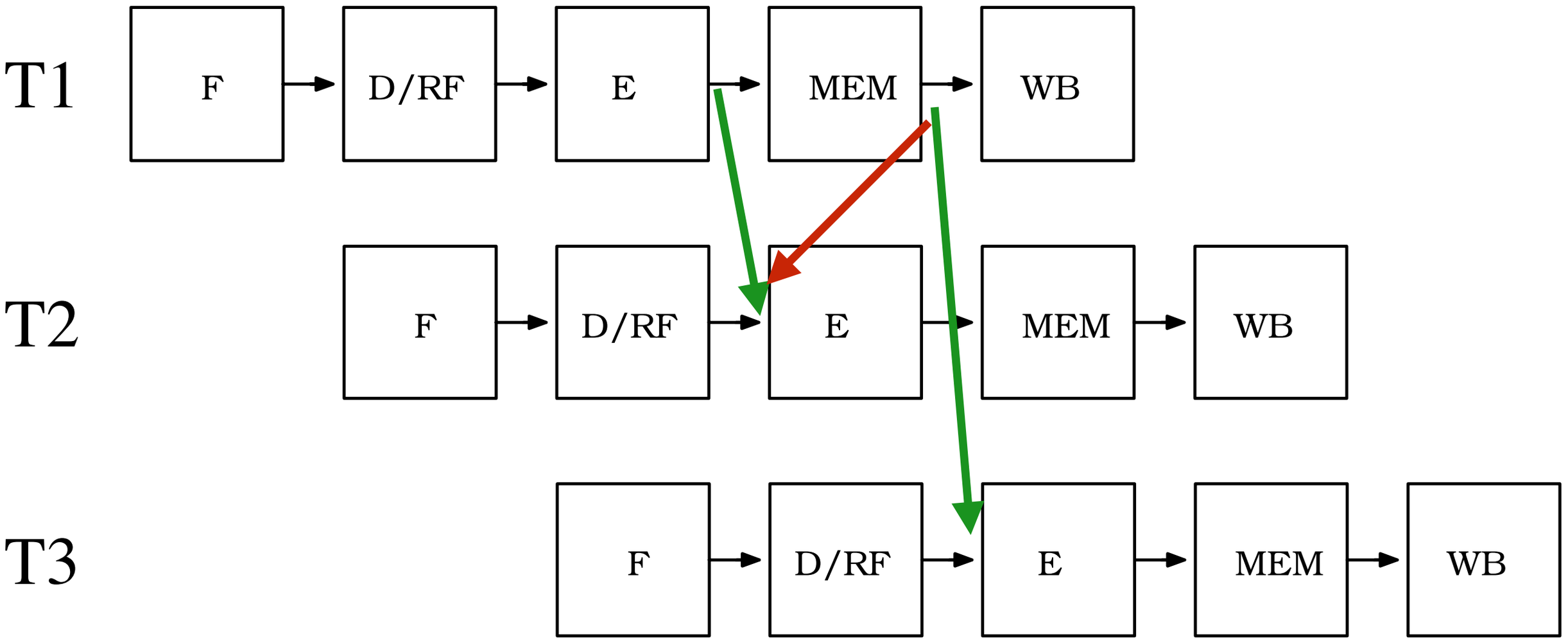


Avoiding all delays?

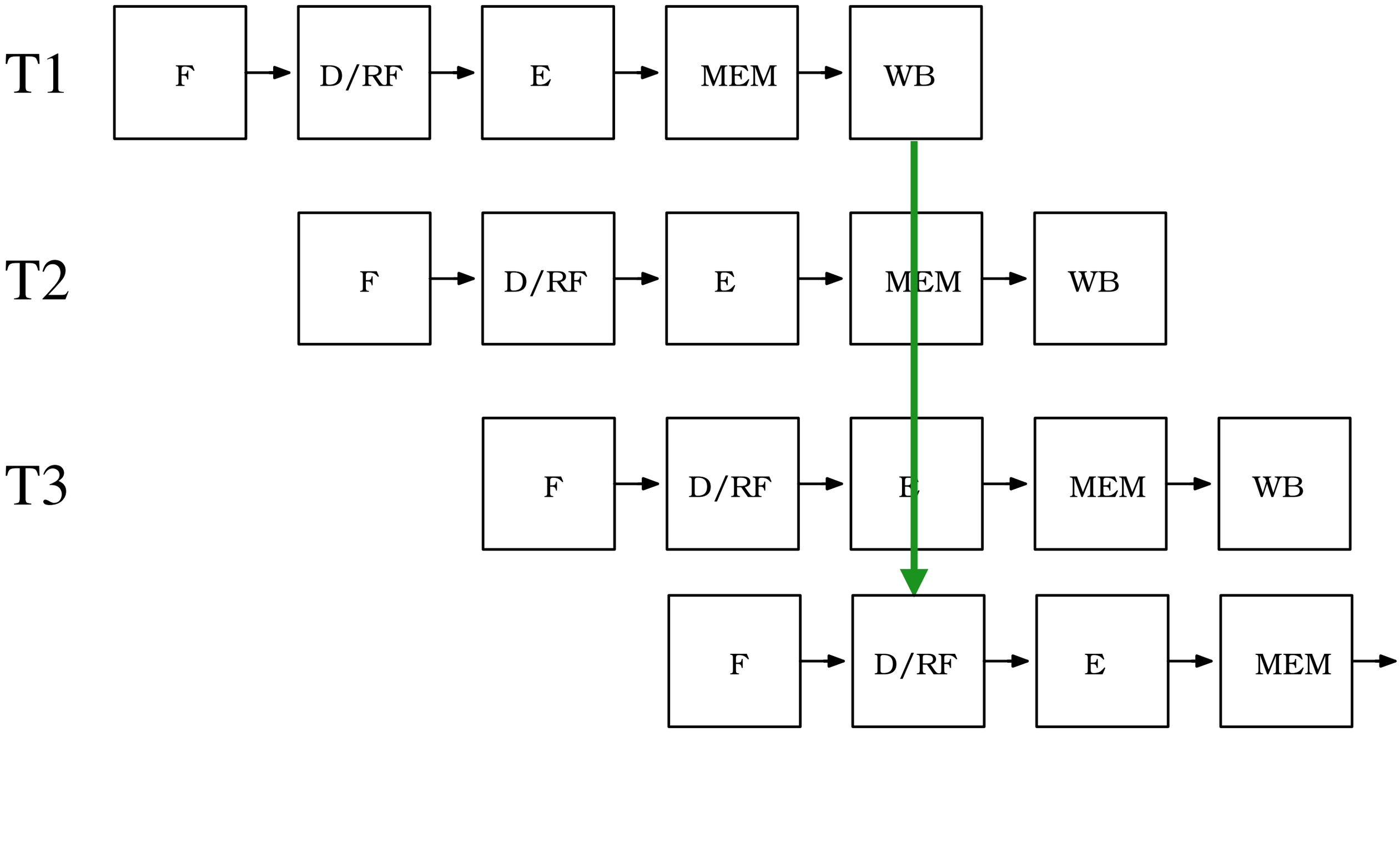
LOAD R1, R2, #20

ADD R3, R1, R7

- Load yields result in MEM stage, which is not available at the time ADD enters EX.



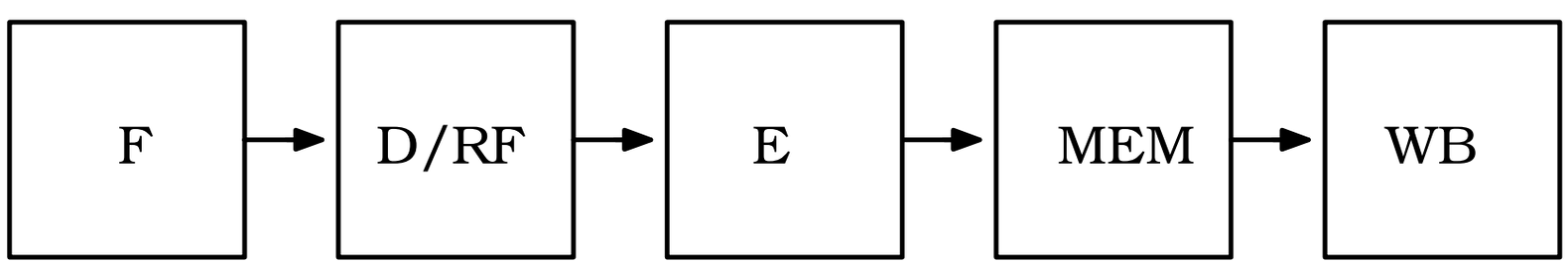
LOAD R1, R2, #20
ADD R3, R1, R7


T1
F
D/RF
E
MEM
WB
T2
F
D/RF
E
MEM
WB
T3
F
D/RF
E
MEM
WB
F
D/RF
E
MEM

Prioritizing forwarding sources

- Choose source closest in program order

AND R1, R2, R3
ADD R1, R1, #1
SUB R4, R1, R5



Software Interlocking

- Arrange instructions so that interlocked instructions are separated by unrelated instructions.
- Avoids stalls due to flow dependencies.
- Applies to instructions with predictable latency.
- Deps on LOAD can be scheduled optimistically.

Requirements for Software Interlocking

- Requires specific details of pipeline
- Implemented as postpass phase of compiler
- Assumes no hardware interlocking
  - D/RF must issue every cycle. Requires Q NOPs for Q cycles of stall.
  - Only possible algorithm, if no hardware interlocking

Outline of Algorithm:

1. Construct the DFG of a basic block of code (**basic block** = code fragment with one entry point and one exit point)

2. Initialization:

(a) Create a list *L* of instructions that can be started when control flows into the basic block – these instructions are the ones whose inputs are all ready and whose output and anti–dependencies have cleared.

(b) Create an output list, *R*, which is initially empty.

(c) Initialize a data structure to keep track of the state of the CPU as follows:

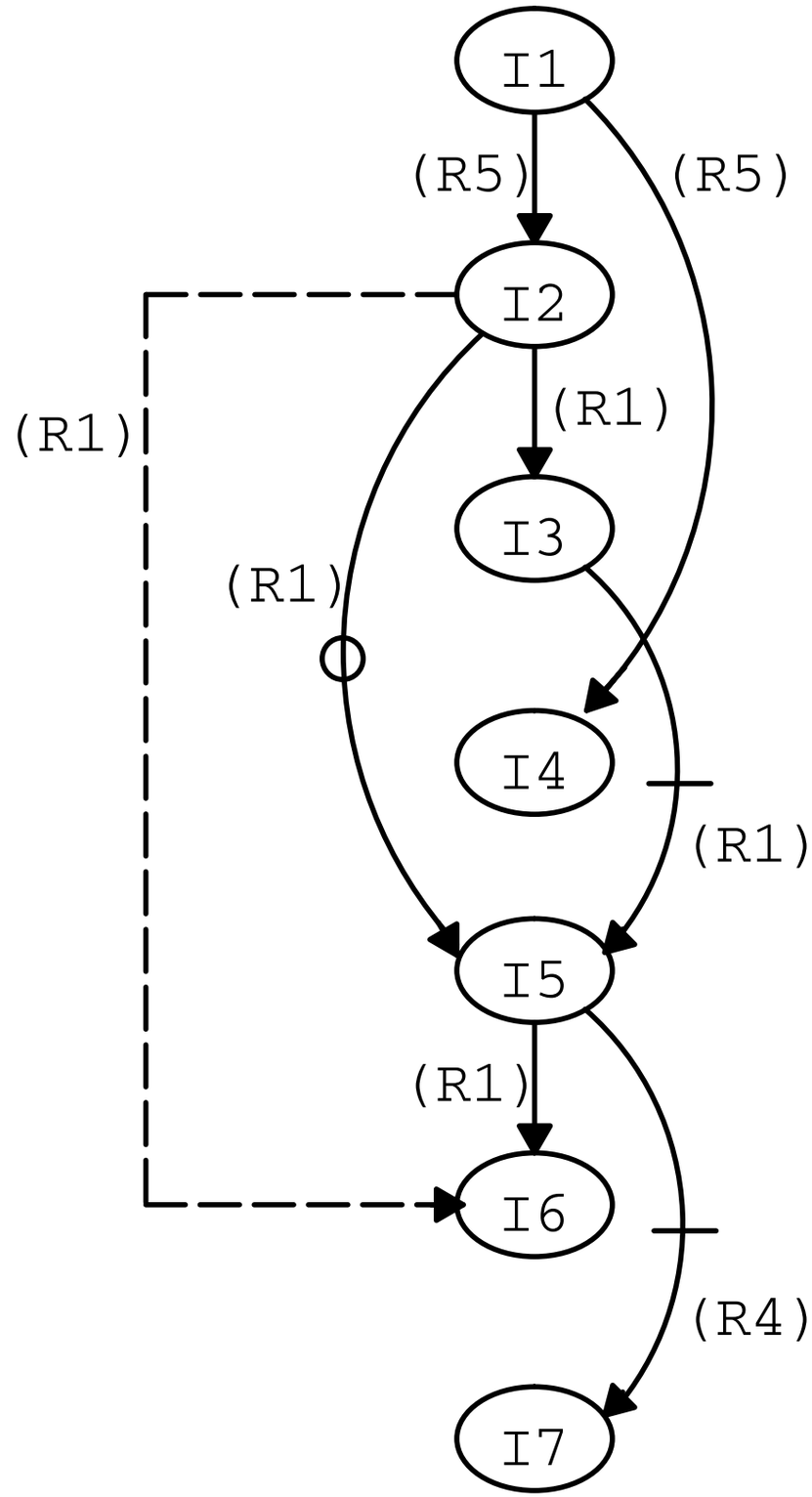
- Mark each register that contains a valid data item as valid
- Mark each FU as free

3. Pretend that an instruction is issued:

(a) If *L* is empty, add a `NOP` to the end of the list *R*

DFG

```
I1:     FMUL    R5, R7, R8
I2:     FADD    R1, R2, R5
I3:     MOV     R9, R1
I4:     FADD    R6, R5, R8
I5:     FDIV    R1, R4, R11
I6:     FST     R1, R10, #100
I7:     FADD    R4, R7, R11
```



(b) If *L* is non–empty, then check if an instruction from *L* can be issued as follows:

For an instruction I in *L* to be issuable:

- All the other instructions that it depends on (for a flow, anti or output dependency) must have completed (i.e., updated the necessary registers or memory locations) or have delivered the data that I is waiting for through a forwarding mechanism, if such a mechanism exists.

- A function unit for executing the instruction must be free

(i) If there is more than one instruction in *L* that can be issued, some secondary criterion may be used to pick one for issue. The selected instruction is removed from *L* and added to the end of *R*. The status of the destination register of this instruction is marked as busy

(ii) If no instruction from *L* can be issued, add a `NOP` to the end of *R*.

4. Assume that a single cycle has elapsed since the initiation of the instruction that was just added to *R*. Update the state of the processor (FU status, register status etc.) at this point in time.

5. Update *L* based on the newly updated state information:

   – From the DFG, add any instruction to *L* which can now be issued since previously initiated instruction(s) that it depended on have just completed or forwarded any required data

   – Any type of dependence (flow, anti– or output) have to be considered

   – From *L* take out any instruction that cannot be issued because the required FU is busy

6. Go to step 3, looping through steps 3 through 6 till all instruction nodes in the original DFG have been added to *R*

   The final contents of *R*, taken in sequence, gives a software interlocked version of the code for the original basic block

– This rearranged code has the same DFG as the original code
– The rearranged code may also have additional NOPs

- Generally, software interlocking cannot be used to take care of dependencies from a LOAD to another instruction.
  - This is because of the unpredictable timing of a LOAD (cache miss service times can vary): hardware mechanisms are very much required to cope with these dependencies.

An Example:

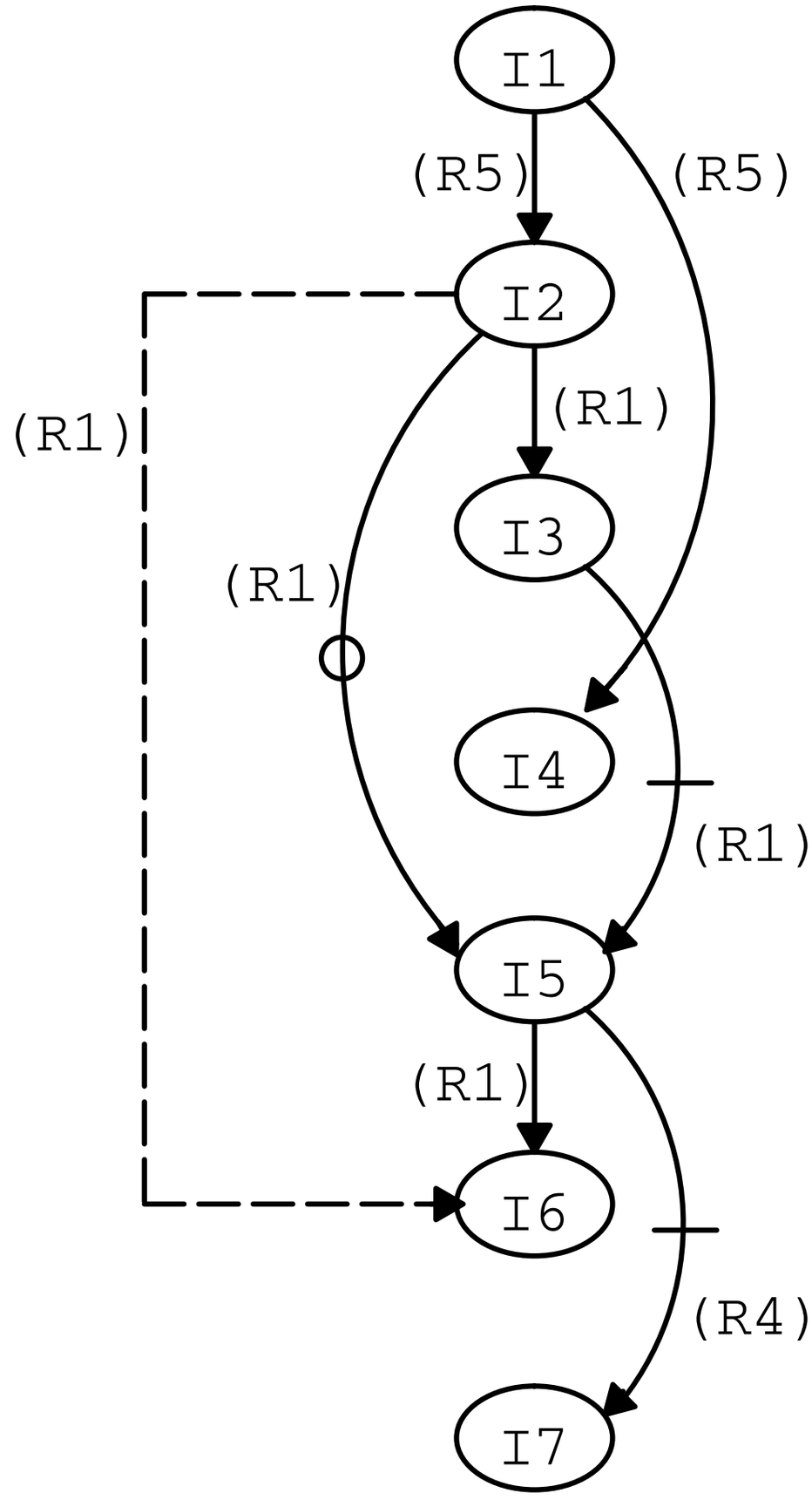
Consider the following code for a basic block

```
/* R2, R4, R5, R7, R8, R10 and R11 contain valid data at this point */


I1:      FMUL     R5, R7, R8
I2:      FADD     R1, R2, R5
I3:      MOV      R9, R1   /* R9 <- R1; done using integer unit */
I4:      FADD     R6, R5, R8
I5:      FDIV     R1, R4, R11
I6:      FST      R1, R10, #100    /* Mem [R6 + 100] <-- R1; dedicated
                                          unit  to compute address */
I7:      FADD     R4, R7, R11
```

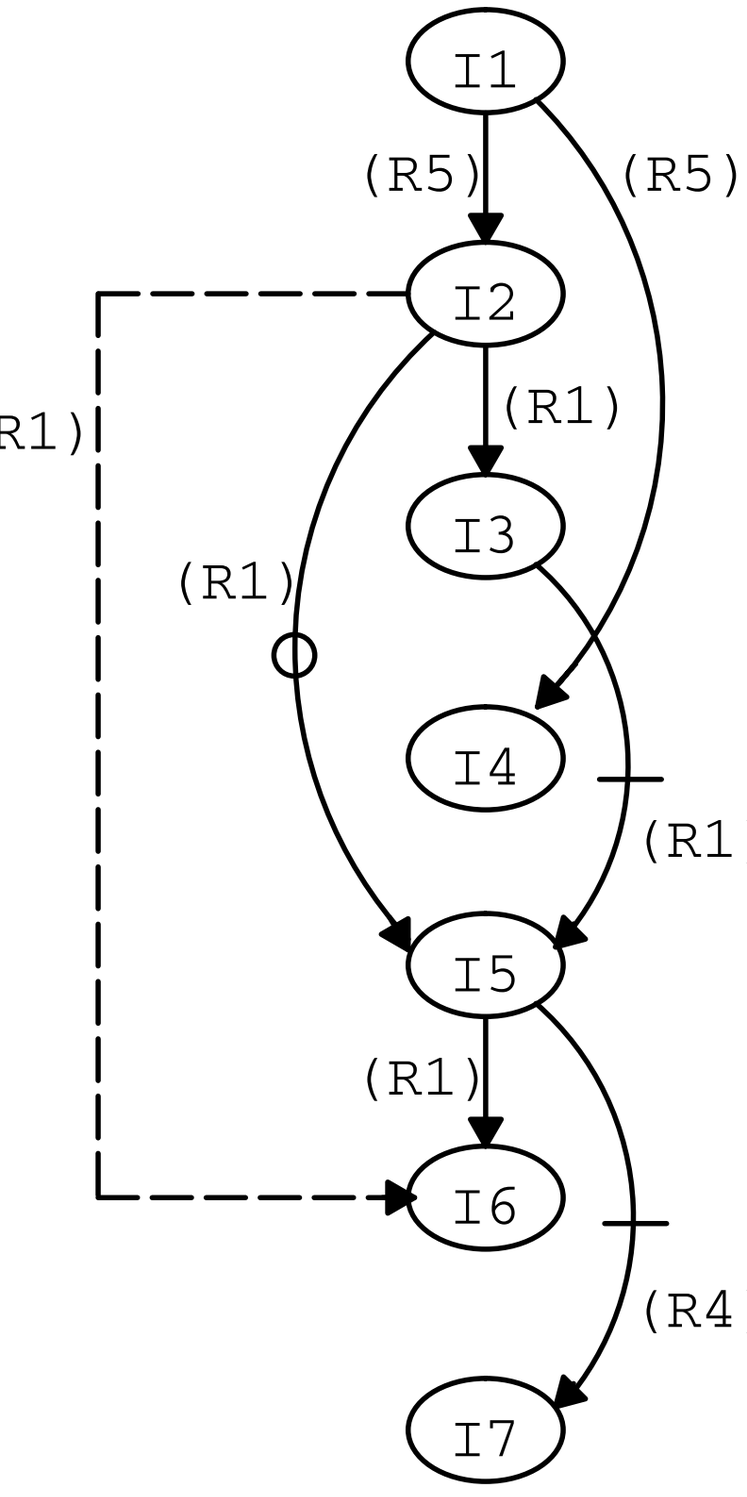
Latencies: FADD: 1 cycle; FMUL: 3 cycles; FDIV: 4 cycles; MOV: 1: FST: 1 cycle for address computation

- DFG for the example:

```
I1:      FMUL    R5, R7, R8
I2:      FADD    R1, R2, R5
I3:      MOV     R9, R1
I4:      FADD    R6, R5, R8
I5:      FDIV    R1, R4, R11
I6:      FST     R1, R10, #100
I7:      FADD    R4, R7, R11
```



```
I1:     FMUL    R5, R7, R8
I2:     FADD    R1, R2, R5
I3:     MOV     R9, R1
I4:     FADD    R6, R5, R8
I5:     FDIV    R1, R4, R11
I6:     FST     R1, R10, #100
I7:     FADD    R4, R7, R11
```



Pass #	L (at start of pass)	L (at end of pass)	Instruction Added to R	Comments
1	L = {I1}	L = { }	I1	I1 “issued”
2	L = { }	L = { }	NOP	NOP “issued”
3	L = { }	L = { }	NOP	NOP “issued”
4	L = {I2, I4}	L = {I4}	I2	Flow dependency from I1 to I2 satisfied through forwarding I2 “issued” since it has more instructions dependent on it than I4
5	L = {I4, I3}	L = {I4 }	I3	Flow dependency from I2 to I3 satisfied through forwarding; I3 “issued” in preference over I4 With I3’s issue, anti–dependency from I3 to I5 is satisfied.
6	L = {I4, I5}	L = {I4}	I5	I2 updates R1 – this satisfies output dep. to I5, making I5 issuable*; I5 “issued” With I5’s issue, anti–dep. to I7 is satisfied
7	L = {I4, I7}	L = {I7}	I4	I4 “issued”; I4 chosen arbitrarily over I7
8	L = {I7}	L = { }	I7	I7 “issued”
9	L = { }	L = { }	NOP	NOP “issued”
10	L = {I6 }	L = { }	I6	I5 completes, forwarding result to I6; I6 “issued”

* actually satisfied earlier since writes are serialized

- The re–arranged, software–interlocked version of the code, as produced by the algorithm is thus:

```
I1:   FMUL  R5, R7, R8
      NOP
      NOP
I2:   FADD  R1, R2, R5
I3:   MOV   R9, R1
I5:   FDIV  R1, R4, R11
I4:   FADD  R6, R5, R8
I7:   FADD  R4, R7, R11
      NOP
I6:   FST   R1, R10, #100
```

- The re–arranged, software–interlocked version of the code, as produced by the algorithm is thus:

```
I1:   FMUL  R5, R7, R8
      NOP
      NOP
I2:   FADD  R1, R2, R5
I3:   MOV   R9, R1
I5:   FDIV  R1, R4, R11
I4:   FADD  R6, R5, R8
I7:   FADD  R4, R7, R11
      NOP
I6:   FST   R1, R10, #100
```

Old version

```
I1:     FMUL    R5, R7, R8
I2:     FADD    R1, R2, R5
I3:     MOV     R9, R1
I4:     FADD    R6, R5, R8
I5:     FDIV    R1, R4, R11
I6:     FST     R1, R10, #100
I7:     FADD    R4, R7, R11
```

Limitations on software interlocking

- Code produced is specific to a particular CPU implementation -- no binary compatibility.
- Does not handle variable latencies of LOADs.
  - Software interlocking for best-case latency
  - Hardware to stall on misses
  - Hybrid solutions

Software Pipelining in loops

- Modify loop body by splicing in instructions from consecutive loop iterations
- Write DFG for multiple consecutive iterations; apply software interlocking algorithm.
- Modified loop body has instructions from different loop iterations within new loop body.

- The construction starts by picking up the instruction(s) that has (have) the most impact on latency as the initial member of *L*

- A loop priming code sequence is needed to start the modified loop

- A similar flush code sequence is needed following the modified loop body

- Often, when operation latencies are large, the modified loop is obtained by starting with an unrolled version of the loop: this requires more registers

Example:

Consider the nested loops used for multiplying two–dimensional matrices:

```
for (i = 0; i++; i < N)
    for (j = 0; j++; j < N)
        for (k = 0; k++; k < N)
            c[i,j] = C[i,j] + A[i,k] * B[k, j]
```

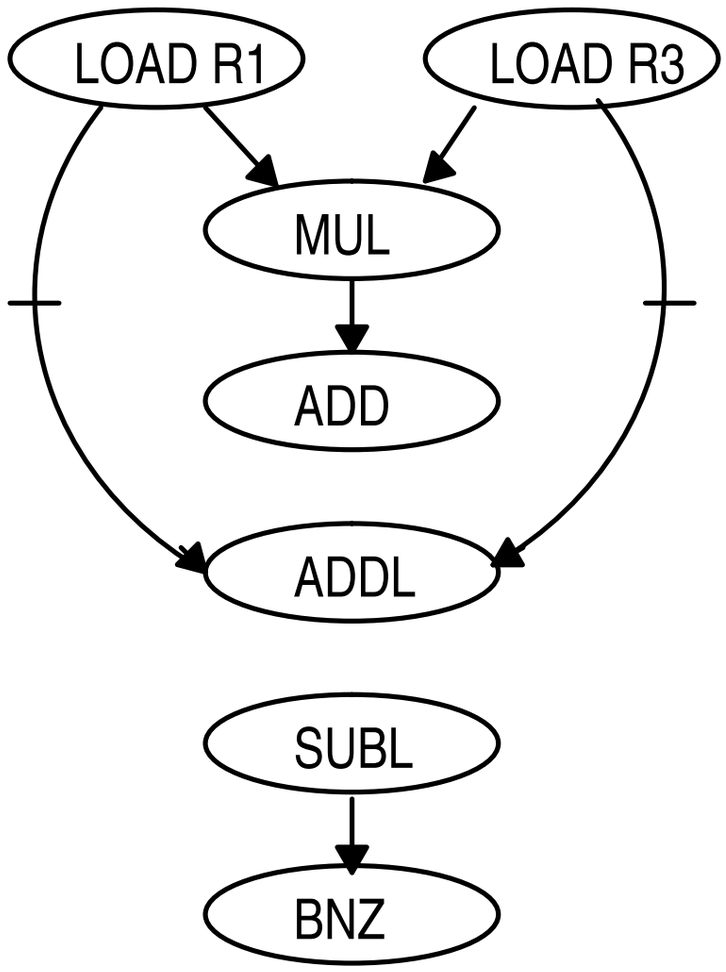
– assume that the array C[ ] is initialized to zeros.

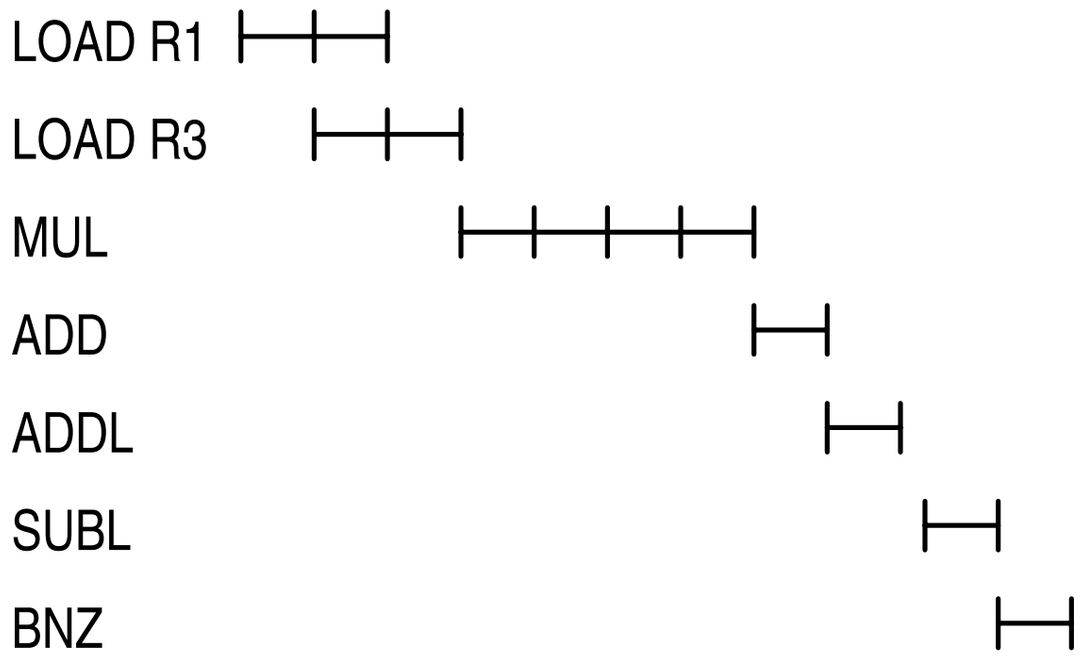
An straightforward implementation of the body of the inner loop in APEX is:

```
loop:  LOAD   R1, R2, R6    /*  load A[i,k] into R1: R1 <- A[i,k] */
       LOAD   R3, R2, R7    /*  R3 <- B[k,j], B is stored transposed */
       MUL    R4, R1, R3    /*  form A[i,k] * B[k,j] */
       ADD    R5, R4, R5    /* C[i,j]in R5; C[i,j] += A[I,k] * B[k,j] */
       ADDL   R2, R2, #4    /* increment k */
       SUBL   R9, R9, #1    /* decrement inner loop counter */
       BNZ loop
```

LOAD: 2 cycles
MUL: 4 cycles
all others: 1 cycle

The DFG for the loop body is:





Gantt chart showing execution latencies

This DFG *does not* show dependencies from one iteration to a subsequent one (“loop–carried dependencies”)

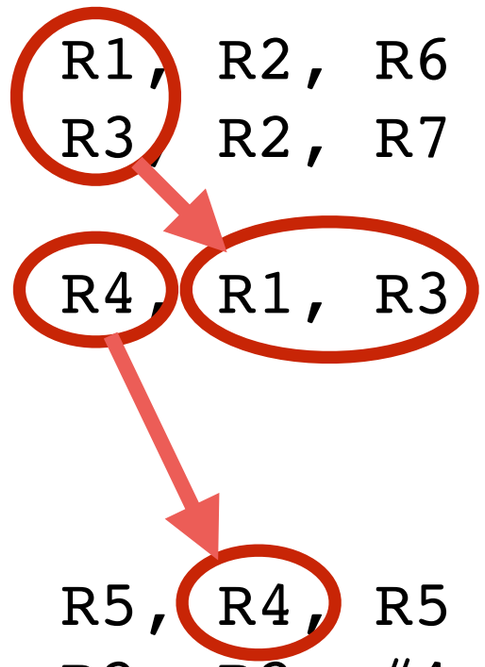
The software pipelined version that reduces the interlocking delays is:

```
    {code to prime loop}
loop:  MULR4, R1, R3    /* perform the multiplication for iteration k */
    LOAD   R1, R2, R6    /* load A[i, k+1] for iteration k+1 */
    LOAD   R3, R2, R7    /* load B[k+1, j] for iteration k+1 */
    ADDL   R2, R2, #4    /* increment for iteration k+1 */
    ADDR5, R5, R4     /* update C[i, j] for iteration k */
    SUBL   R9, R9, #1
    BNZloop
{code to flush loop}
```

```
0: LOAD  R1, R2, R6
1: LOAD  R3, R2, R7
2: MUL   R4, R1, R3
3: ADD   R5, R4, R5
4: ADDL  R2, R2, #4
5: SUBL  R9, R9, #1
```

```
0: LOAD   R1, R2, R6
1: LOAD   R3, R2, R7
2: MUL    R4, R1, R3
3: ADD    R5, R4, R5
4: ADDL   R2, R2, #4
5: SUBL   R9, R9, #1
```

```
0: LOAD   R1, R2, R6
1: LOAD   R3, R2, R7
stall
2: MUL    R4, R1, R3
stall
stall
stall
3: ADD    R5, R4, R5
4: ADDL   R2, R2, #4
5: SUBL   R9, R9, #1
```



```
0: LOAD  R1, R2, R6
1: LOAD  R3, R2, R7
stall
2: MUL   R4, R1, R3
stall
stall
stall
3: ADD   R5, R4, R5
4: ADDL  R2, R2, #4
5: SUBL  R9, R9, #1
6: BNZ Loop
```

```
0: LOAD  R1, R2, R6
1: LOAD  R3, R2, R7
stall
2: MUL   R4, R1, R3
stall
stall
stall
3: ADD   R5, R4, R5
4: ADDL  R2, R2, #4
5: SUBL  R9, R9, #1
6: BNZ Loop
```

```
A: LOAD  R1, R2, R6
B: LOAD  R3, R2, R7
stall
C: MUL   R4, R1, R3
stall
stall
stall
D: ADD   R5, R4, R5
E: ADDL  R2, R2, #4
F: SUBL  R9, R9, #1
G: BNZ Loop
```

```
0: LOAD  R1, R2, R6
1: LOAD  R3, R2, R7
stall
2: MUL   R4, R1, R3
stall
stall
stall
3: ADD   R5, R4, R5
4: ADDL  R2, R2, #4
5: SUBL  R9, R9, #1
6: BNZ Loop
```

```
0: LOAD  R1, R2, R6
1: LOAD  R3, R2, R7
stall
2: MUL   R4, R1, R3
stall
stall
stall
3: ADD   R5, R4, R5
4: ADDL  R2, R2, #4
5: SUBL  R9, R9, #1
6: BNZ Loop
A: LOAD  R1, R2, R6
B: LOAD  R3, R2, R7
stall
C: MUL   R4, R1, R3
stall
stall
stall
D: ADD   R5, R4, R5
E: ADDL  R2, R2, #4
F: SUBL  R9, R9, #1
G: BNZ Loop
```

```
0: LOAD  R1, R2, R6
1: LOAD  R3, R2, R7
stall
2: MUL   R4, R1, R3
stall
stall
stall
3: ADD   R5, R4, R5
4: ADDL  R2, R2, #4
5: SUBL  R9, R9, #1
6: BNZ Loop
```

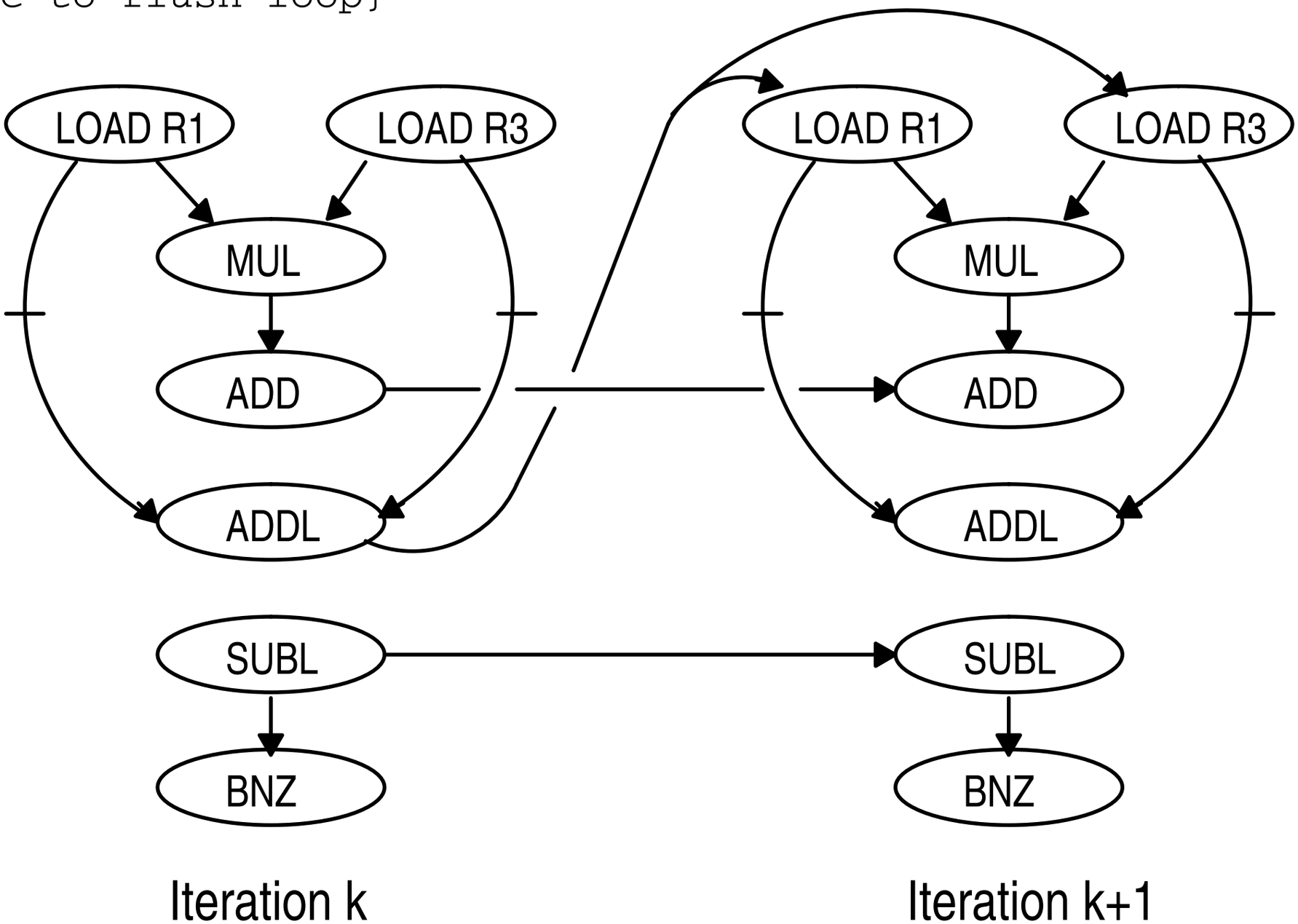
```
0: LOAD  R1, R2, R6
1: LOAD  R3, R2, R7
stall
2: MUL   R4, R1, R3
stall
stall
stall
3: ADD   R5, R4, R5
4: ADDL  R2, R2, #4
5: SUBL  R9, R9, #1
6: BNZ Loop

A: LOAD  R1, R2, R6
B: LOAD  R3, R2, R7
stall
C: MUL   R4, R1, R3
stall
stall
stall
D: ADD   R5, R4, R5
E: ADDL  R2, R2, #4
F: SUBL  R9, R9, #1
G: BNZ Loop
```

```
0: LOAD  R1, R2, R6
1: LOAD  R3, R2, R7
4: ADDL  R2, R2, #4
```

```
2: MUL   R4, R1, R3
A: LOAD  R1, R2, R6
B: LOAD  R3, R2, R7
E: ADDL  R2, R2, #4
3: ADD   R5, R4, R5
5: SUBL  R9, R9, #1
6: BNZ Loop
```

```
C: MUL   R4, R1, R3
stall
stall
stall
D: ADD   R5, R4, R5
~~F: SUBL  R9, R9, #1~~
~~G: BNZ Loop~~
```

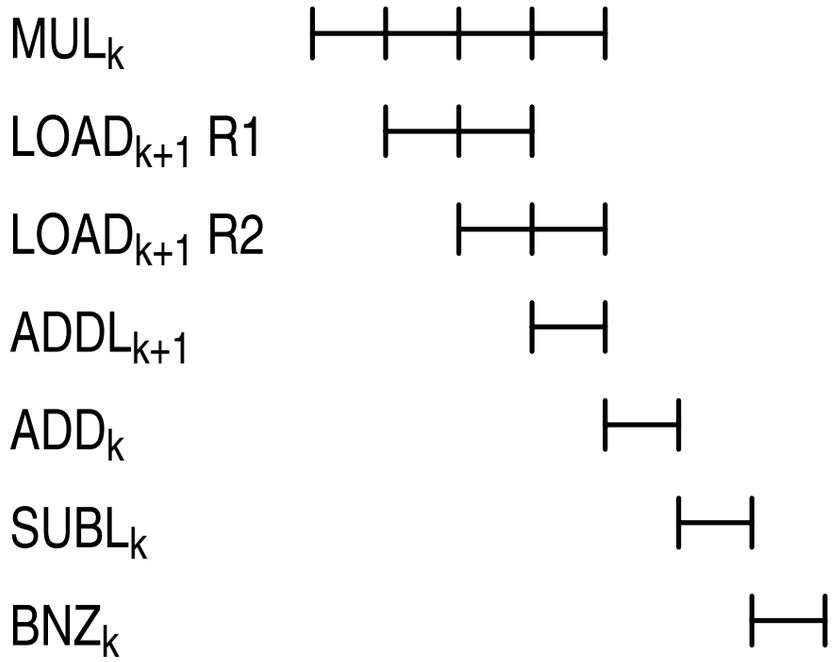
```
    {code to prime loop}
loop: MUL R4, R1, R3    /* perform the multiplication for iteration k */
      LOAD  R1, R2, R6    /* load A[i, k+1] for iteration k+1 */
      LOAD  R3, R2, R7    /* load B[k+1, j] for iteration k+1 */
      ADDL  R2, R2, #4    /* increment for iteration k+1 */
      ADD R5, R4, R4    /* update C[i, j] for iteration k */
      SUBL  R9, R9, #1
      BNZ loop
    {code to flush loop}
```



The final schedule produced is:

$$R = \{ MUL_k, LOAD_{k+1}\ R1, LOAD_{k+1}\ R2, ADDL_{k+1}, ADD_k, SUB_k, BNZ_k \}$$

This modified loop – the **software pipelined loop**, as mentioned earlier, takes only 7 cycles per iteration:



```
loop:  LOAD   R1, R2, R6    /*  load A[i,k] into R1: R1 <- A[i,k] */
       LOAD   R3, R2, R7    /*  R3 <- B[k,j], B is stored transposed */
       MUL    R4, R1, R3    /*  form A[i,k] * B[k,j] */
       ADD    R5, R4, R5    /* C[i,j]in R5; C[i,j] += A[I,k] * B[k,j] */
       ADDL   R2, R2, #4    /* increment k */
       LOAD   R10, R2, R6      /*  load A[i,k+1] into R10 */
       LOAD   R11, R2, R7      /*  R11 <- B[k+1,j] */
       MUL    R12, R10, R11    /*  form A[i,k+1] * B[k+1,j] */
       ADD    R5, R12, R5   /* C[i,j]in R5; C[i,j] += A[I,k] * B[k,j] */
       ADDL   R2, R2, #4    /* increment k */
       SUBL   R9, R9, #1    /* decrement inner loop counter */
       BNZ loop
```

```
loop:  LOAD   R1, R2, R6
       LOAD   R3, R2, R7
       MUL    R4, R1, R3
       ADD    R5, R4, R5
       ADDL   R2, R2, #4
       LOAD   R10, R2, R6
       LOAD   R11, R2, R7
       MUL    R12, R10, R11
       ADD    R5, R12, R5
       ADDL   R2, R2, #4
       SUBL   R9, R9, #1
       BNZ loop
```

```
loop:  MUL    R4, R1, R3
       MUL    R12, R10, R11
       LOAD   R1, R2, R6
       LOAD   R3, R2, R7
       ADDL   R2, R2, #4
       LOAD   R10, R2, R6
       LOAD   R11, R2, R7
       ADDL   R2, R2, #4
       ADD    R5, R4, R5
       ADD    R5, R12, R5
       SUBL   R9, R9, #1
       BNZ loop
```

The main limitations of static instruction scheduling techniques for coping with dependencies, such as software interlocking and software pipelining were noted earlier:

- Software scheduling does not allow binary compatibility
- Software scheduling does not cope efficiently with FUs that have unpredictable latencies
- Software scheduling techniques can add NOPs that can increase the size of the binaries
- Software scheduling techniques cannot cope with dependencies when program branching occurs *unless predictions can be made about the branch directions statically*
- Does not handle dependencies over memory locations efficiently

Dynamic instruction scheduling

- Manage dependencies automatically
- Out-of-order startup (Issue Queues)
- Out-of-order completion
- Multiple pipelined functional units

Disadvantages of Dynamic Scheduling

- Substantial additional hardware
  - IQ, comparators, LSQ, RoB
  - More power
  - Increased cycle time, increased latency
  - More processor state

Terminology Related to Dynamic Instruction Scheduling

- **Resolving dependencies**: noting and setting up data flow paths to satisfy flow dependencies and, possibly, other types of dependencies as well.

- **Satisfying dependencies**: completion of flow of data to satisfy flow dependency and, possibly, other types of dependencies.

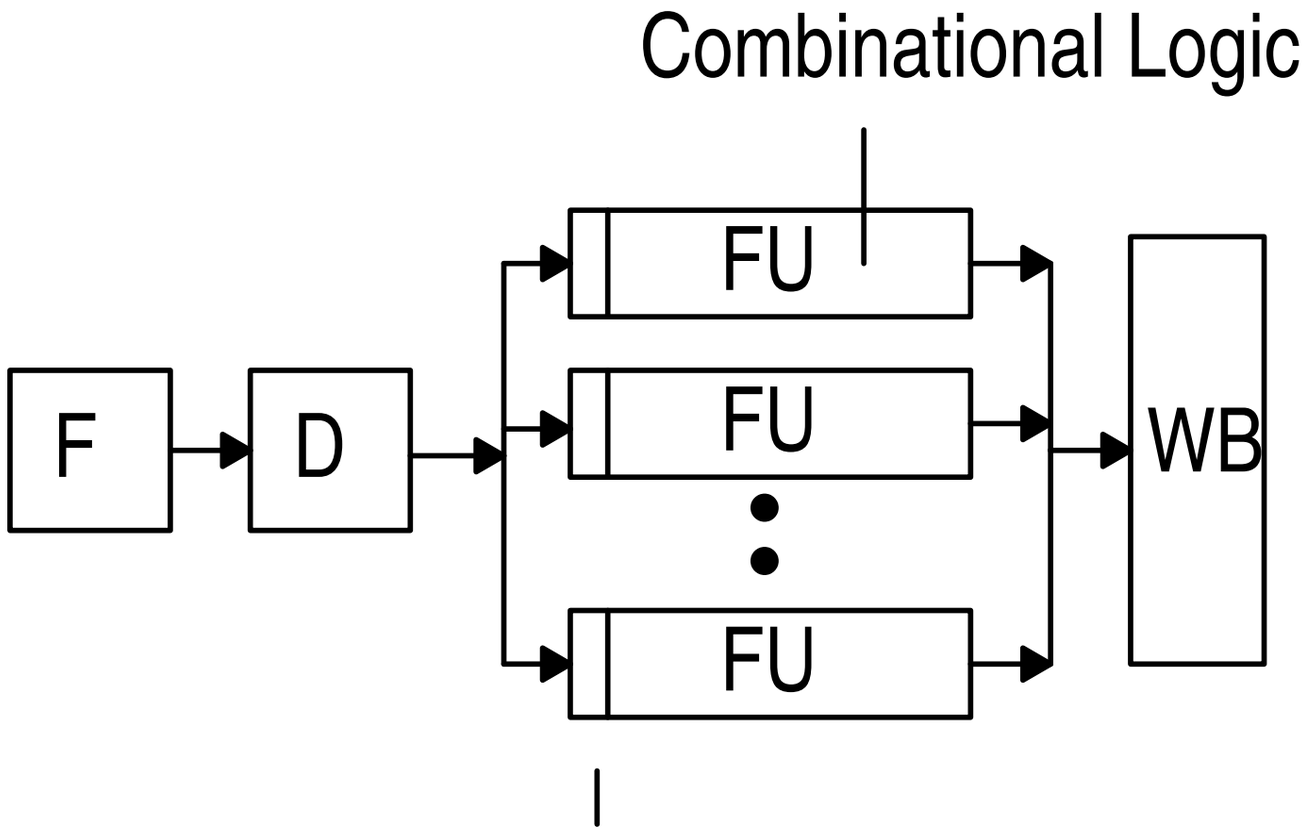
- **Instruction dispatching**: A step that involves decoding one (or more) instruction(s), determining type(s) of VFU needed and resolving dependencies.

- **Instruction issuing**: this refers to the satisfacton of dependencies for an already dispatched instruction, which enables the instruction for execution. The actual execution can be delayed pending the availability of a *physical* FU.

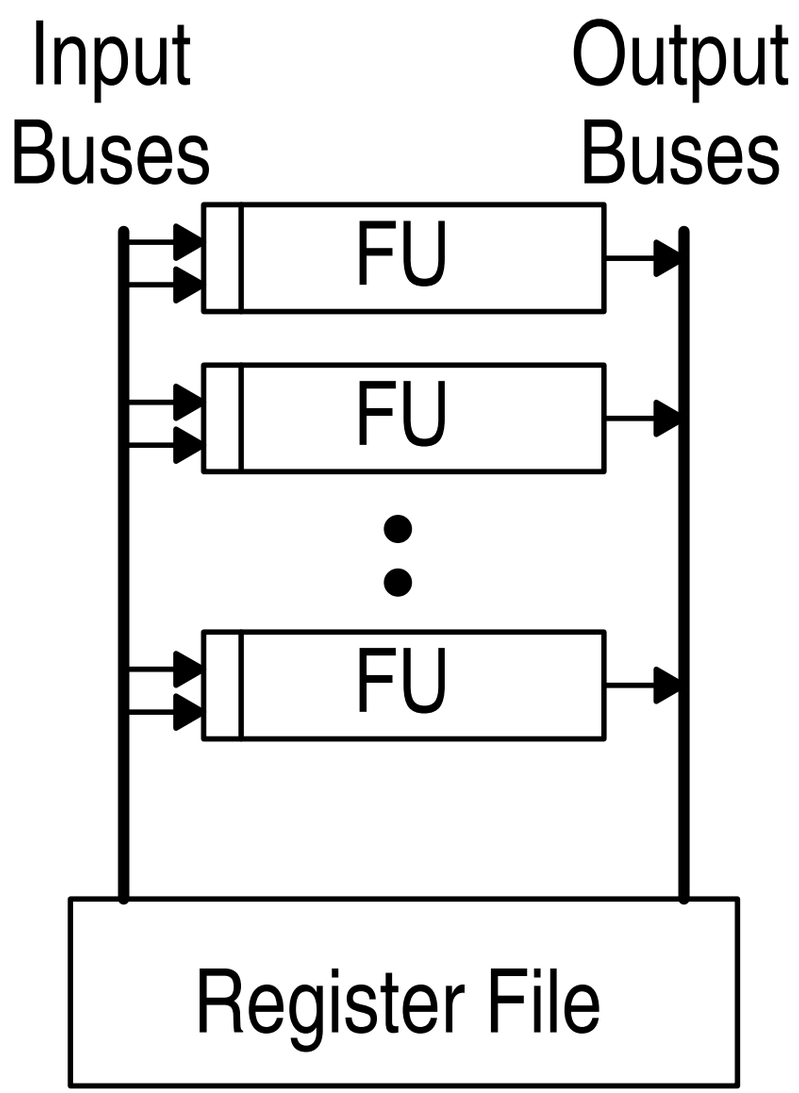
The CDC 6600 Scoreboard Mechanism

- One of the earliest dynamic instruction scheduling techniques was used in the CDC 6600 which:
  - was the fastest supercomputer at one time
  - had a clean RISCy ISA (with some exceptions)
  - survived into the early 80s
  - Had 16 FUs (5 load/store FUs, 7 integer FUs, 4 floating pt. FUs)

We will illustrate this mechanism by adapting it for APEX, preserving all the main features of the original mechanism:



Pipeline



Register Operand Flow

Scoreboard

- Central structure that tracks all instructions and register contents
- Controls:
  - Dispatch
  - Transfer of operands from RF to FUs
  - Writeback

- The scoreboard does this by maintaining the following status information, which is updated continuously:

  - Function Unit Status:
    - allocation status: busy or free
    - operation type (e.g., addition, subtraction)
    - source register addresses
    - destination register address
    - status of source registers (busy or free)

  - Register Status:
    - ids of function units that are computing the data for the source registers (if applicable)

  - Instruction Status: maintained for each *active* instruction
    - Pipeline stage where instruction is currently in and its status.

CDC6600 Dispatch Rule

- Can dispatch from Decode when:
  - FU of required type is free
  - No active or dispatched instruction has same dest register
- Don't need source operands to be available

CDC6600 Startup Rule

- Requirements:
  - Must have **all** input operands valid
    - Required input bus is free
- Enforces flow dependencies
- Optimization: When input operands are available, dispatch and RF read simultaneous

CDC6600 Completion Rule

- FU holds result until scoreboard requests writeback.
- Writeback can proceed when no FU waiting on earlier version of dest register.
- Allowing only one out-standing write to a given register (dispatch rule) simplifies bookkeeping.

To summarize, the scoreboard logic enforces the 3 types of dependencies as follows:

Dependency Type	How Enforced
Flow	Startup rule
Anti	Completion rule
Output	Dispatch rule

FU	Latency	Comments
Integer	1	Implements integer ops including MOVC
Floating Point Add	4	FP adds & subtracts
Floating Point Mul.	3	FP multiplication
Floating Point Div.	7	FP division
Load unit	3*	Integer & FP loads; Latency includes time for address computation
Store unit	3*	Integer & FP stores; Latency includes time for address computation

Consider now the execution of the following code fragment on this CPU. The source register *status* fields in the scoreboard entries of FUs are initialized as invalid (busy), while the status fields of the destination register in the FU entry are marked as free. (*Verify why this makes sense!*). FUs are also marked as free during initialization.

For the register status, the status fields are initialized as busy.

```
(I0)   MOVC R1, #200          /* Implemented on IntFU */
(I1)   FLOAD F0, R1, #0
(I2)   FLOAD F1, R1, #8
(I3)   FDIV F2, F1, F0
(I4)   FMUL F3, F1, F0
(I5)   FSTORE F2, R1, #16
(I6)   ADDL R1, R1, #400
(I7)   FSUB F4, F1, F0
```

Here, the Fs refer to floating point registers (which are different from integer registers, designated as Rs)

MOVC R1, #200

We now trace events on a cycle by cycle basis, for the first few cycles to show how the scoreboard entries are updated *at the end* of the cycle, starting with the first dispatch:

Cycle 2: I0 (MOVC) is dispatched in this cycle to the IntFU. The literal (#200) is also moved to the IntFU via one of the input buses.

The scoreboard entries updated at the end of this cycle are:

- Instruction status: I2 is in F; I1 is in D; I0 is in Int FU & executing
- Register status: R1 is busy, expecting an update from the Int FU.
- FU status:

FU id	op type	status	src1/status	src2/status	dest/status
IntFU	movc	busy	XX/XX	XX/XX	R1/busy

XX = don't care

FLOAD F0, R1, #0

Cycle 3: I1 (FLOAD) is dispatched in this cycle to the LoadFU. Its literal (#0) is also moved to the LoadFU. The MOVC is executing on the IntFU. I2 cannot be issued, as the LoadFU is busy.

The scoreboard entries updated at the end of this cycle are:

- Instruction status: I3 is in F; I2 is in D; I1 is in LoadFU; I0 is in Int FU
- Register status: R1 is busy, expecting an update from the Int FU; F0 is busy, expecting an update from the LoadFU.
- FU status:

FU id	op type	status	src1/status	src2/status	dest/status
IntFU	movc	busy	XX/XX	XX/XX	R1/busy-
LoadFU	fload	busy	R1/busy	XX/XX	F0/busy

FLOAD F1, R1, #8

Cycle 4: I2 (the second FLOAD) cannot be dispatched in this cycle (the required FU is busy). At the beginning of this cycle, the Int FU finishes. Also, during this cycle the scoreboard signals the IntFU to write R1.

The scoreboard entries updated in this cycle are:

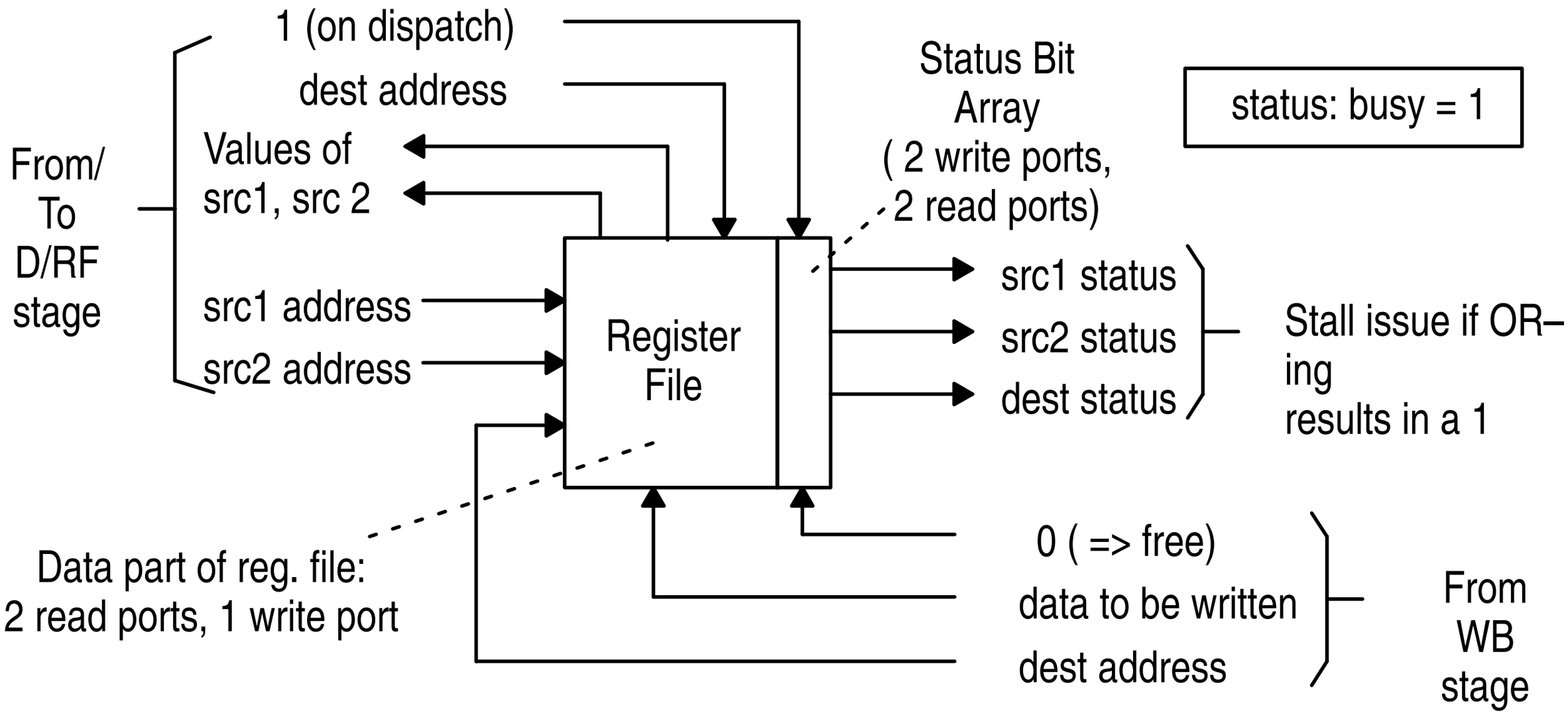
- Instruction status: I3 is in F; I2 is in D; I1 is (still) in LoadFU; I0 is in WB.
- Register status: R1 is free, updated from the Int FU; F0 is busy, expecting an update from the LoadFU.
- FU status:

FU id	op type	status	src1/status	src2/status	dest/status
IntFU	XX	free	XX/XX	XX/XX	XX
LoadFU	fload	busy	R1/free	XX/XX	F0/busy

FLOAD F1, R1, #8

Cycle 5: I2 (the second FLOAD) cannot again be dispatched in this cycle (the required FU is busy). At the beginning of this cycle, the scoreboard signals the LoadFU to read in the value of R1. The scoreboard entries updated at the end of this cycle are:

- Instruction status: I3 is in F; I2 is in D; I1 is in LoadFU.
- Register status: R1 is free; F0 is busy, expecting an update from the LoadFU.
- FU status: Similar as that at the end of cycle 4, except that src1/status field of LoadFU is R1/free.


1 (on dispatch)
dest address
Values of
src1, src 2
From/
To
D/RF
stage
src1 address
src2 address
Register
File
Status Bit
Array
(2 write ports,
2 read ports)
status: busy = 1
src1 status
src2 status
dest status
Stall issue if OR–
ing
results in a 1
Data part of reg. file:
2 read ports, 1 write port
0 (=> free)
data to be written
dest address
From
WB
stage

- Potential improvements to the CDC 6600 scoreboarding mechanism:
  - Allow operands to be transferred to FUs in a piecemeal fashion
  - Incorporate forwarding logic
  - Incorporate write buffers within the FUs – this frees up the input latches for a FU to accept a new set of inputs

Tomasulo's Algorithm

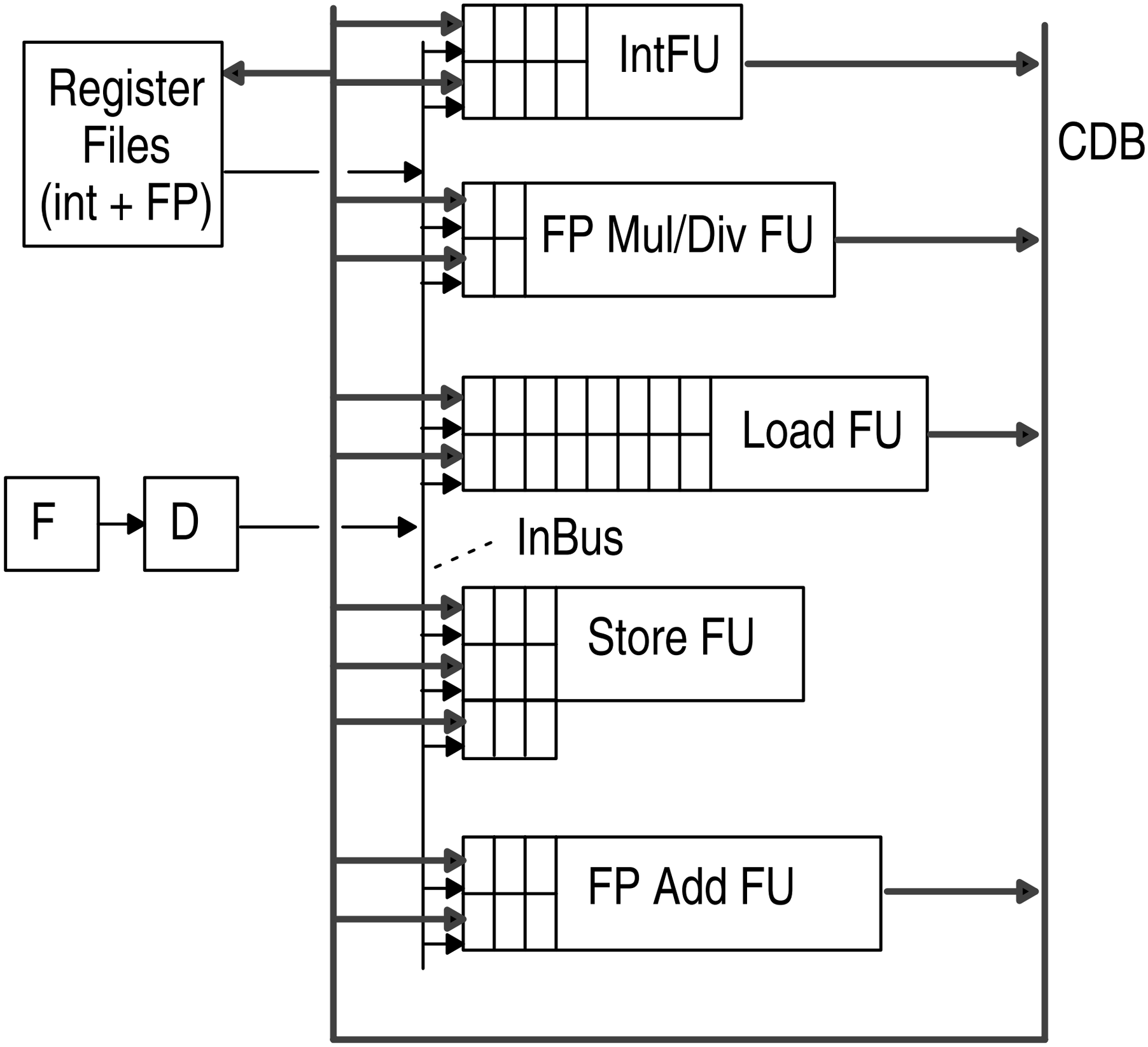
- Employed in the IBM 360, model 91, this scheme is named after its inventor:
  - Landmark in pipeline CPU design
  - Remains sophisticated even by today's standard
  - This dynamic issuing scheme was implemented within the floating point unit of the 360/91
  - The 360/91 was a commercial failure – the 360/85 ran faster using a cache.
  - The Tomasulo scheme was way ahead of its times: variations of this scheme started showing up in microprocessors only in the early 90s!

- The main features of the datapath in APEX using Tomasulo's algorithm are:

  – Multiple FUs as shown. All FUs have associated *reservation stations*

  – A common bus, called the CDB ("common data bus") is used for forwarding data from one FU to all destinations that are waiting for the result within a single cycle.

  – Forwarding is accomplished by using statically assigned **source tags** to identify forwarding sources, as detailed later.


Register
Files
(int + FP)
IntFU
FP Mul/Div FU
Load FU
CDB
F
D
InBus
Store FU
FP Add FU

VFU	Number	Identifier
Integer VFUs	4	1, 2, 3, 4
Load VFUs	8	5 through 12
FP Add VFUs	3	13, 14, 15
FPMulDiv VFUs	2	16, 17

Notice that the Store VFUs are *not* assigned any unique ids, since they do not need to deliver a data to a waiting VFU or a result. The source ids are called **source tags**.

- Note that 5–bit tags suffice for the purpose of unique naming
- Note also that the tag value zero (00000) is unassigned to any source

- The physical FUs are assumed to be **pipelined** and have the following latencies:

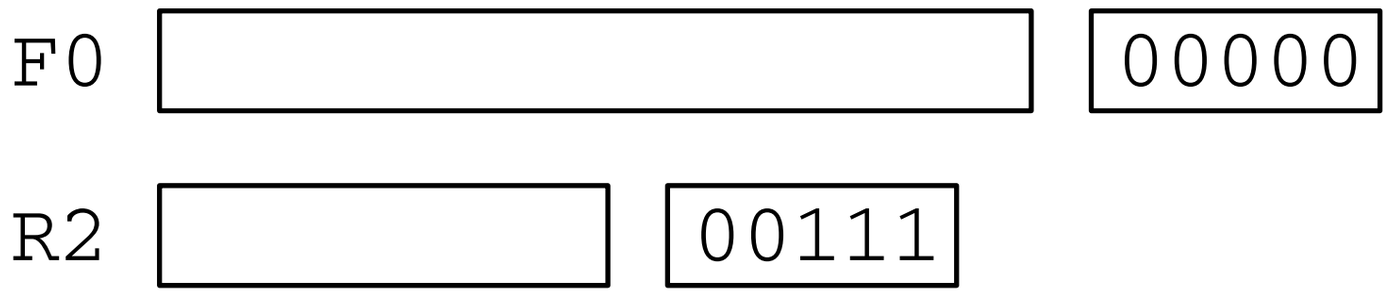
Physical FU	Latency (in cycles)
Integer FU	1
Load FU	3*
Store FU	3*
FP Add FU	4
FP Mul./Div. FU	7

* If memory target is in cache; longer otherwise

– Assume branches to be processed by the F stage itself using some hardware prediction mechanism. This is not discussed further.

- Each architectural register has an associated **tag register**. If the tag register holds a zero, then the register contains a valid data. If the tag register is non–zero, the contents of the tag register names the VFU that is to write a result into the register.

  Example:



FP register F0 contains valid data; Integer register R2 is expecting data
from virtual load unit with the id 7

FP Add FU (physical FU)

1	00000		00000		FPAddVFU0 (Id = 13)
0	00010		00011		FPAddVFU1 (Id = 14)
1	00001		00101		FPAddVFU2 (Id = 15)

right input data

right input tag register

left input data

left input tag register

status (0 = free; 1 = allocated)

- An instruction can wait in a VFU for three reasons:

  - Waiting for the availability of input operands
  - Waiting for the availability of the physical FU (even if it is pipelined).
  - Waiting for access to the CDB

- A virtual FU is enabled for startup when all tag registers in its associated RSE contain 00000.

  – The startup actually occurs when the associated FU selects one of the enabled RSE.

  – The **issue rule** for a VFU is thus:

    (a) Selected VFU must have all inputs available

  – (b) Associated physical FU is free

Forwarding targets

- Architectural registers
- VFU slots

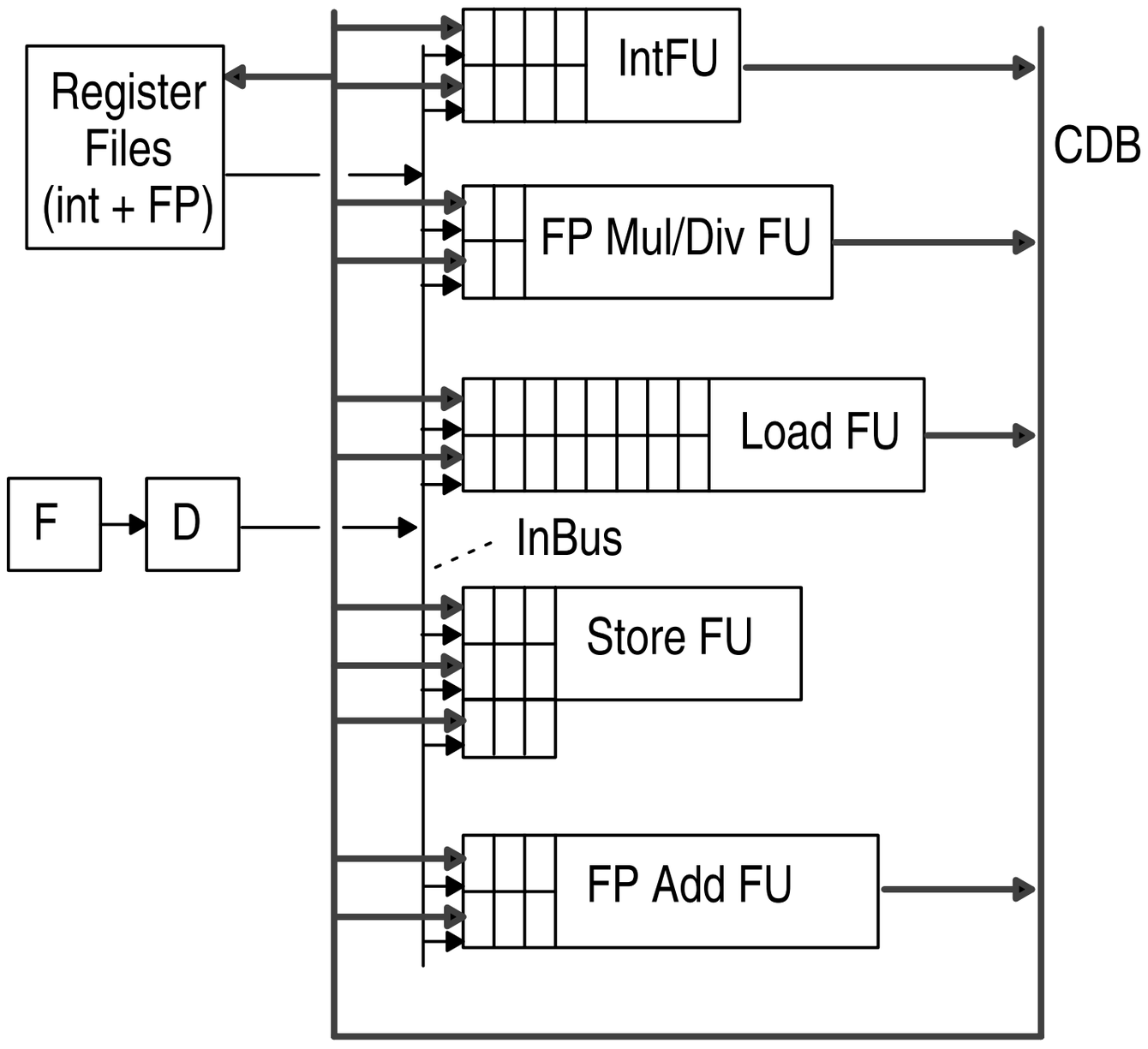
Completion & Forwarding

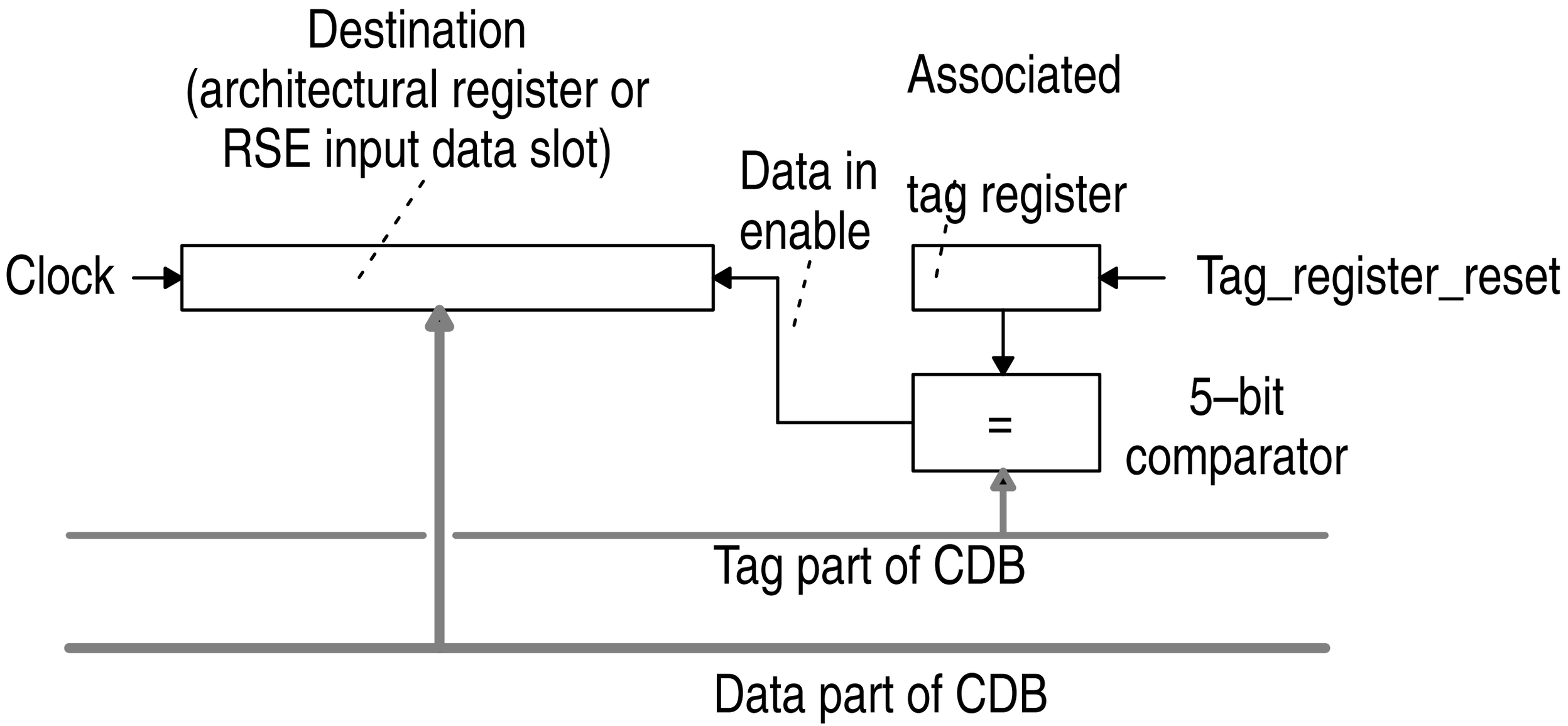
- Completing instructions compete for access to CDB
  - Scheduling and arbitration
- Result broadcast on CDB, along with tag

Satisfying Dependencies

- All destinations continuously monitor CDB, looking for their source tag
- All matching destinations pick up result simultaneously
- Source tag is set to zero after input has been captured

How many comparators?





A tag match enables data from the CDB to be loaded into the destination.
The associated tag register is reset to 00000 after the destination is loaded from the CDB

- **Instruction dispatching rule**: The D stage can dispatch an instruction only if a virtual FU of the required type is free. The following actions are taken on a dispatch depending on the type of instruction:

Dispatching with register source

- Register busy:
  - Architectural reg is tagged with source
    - Copy source tag to VFU
- Register free:
  - Architectural reg has zero tag
  - Copy source data to VFU

Dispatching with literal source

- Copy immediate value into VFU data field
- Set VFU tag field to zero

Updating register with new source tag

- Mark destination register with VFU tag
- Marking busy register with new VFU tag is okay.

Handling completion

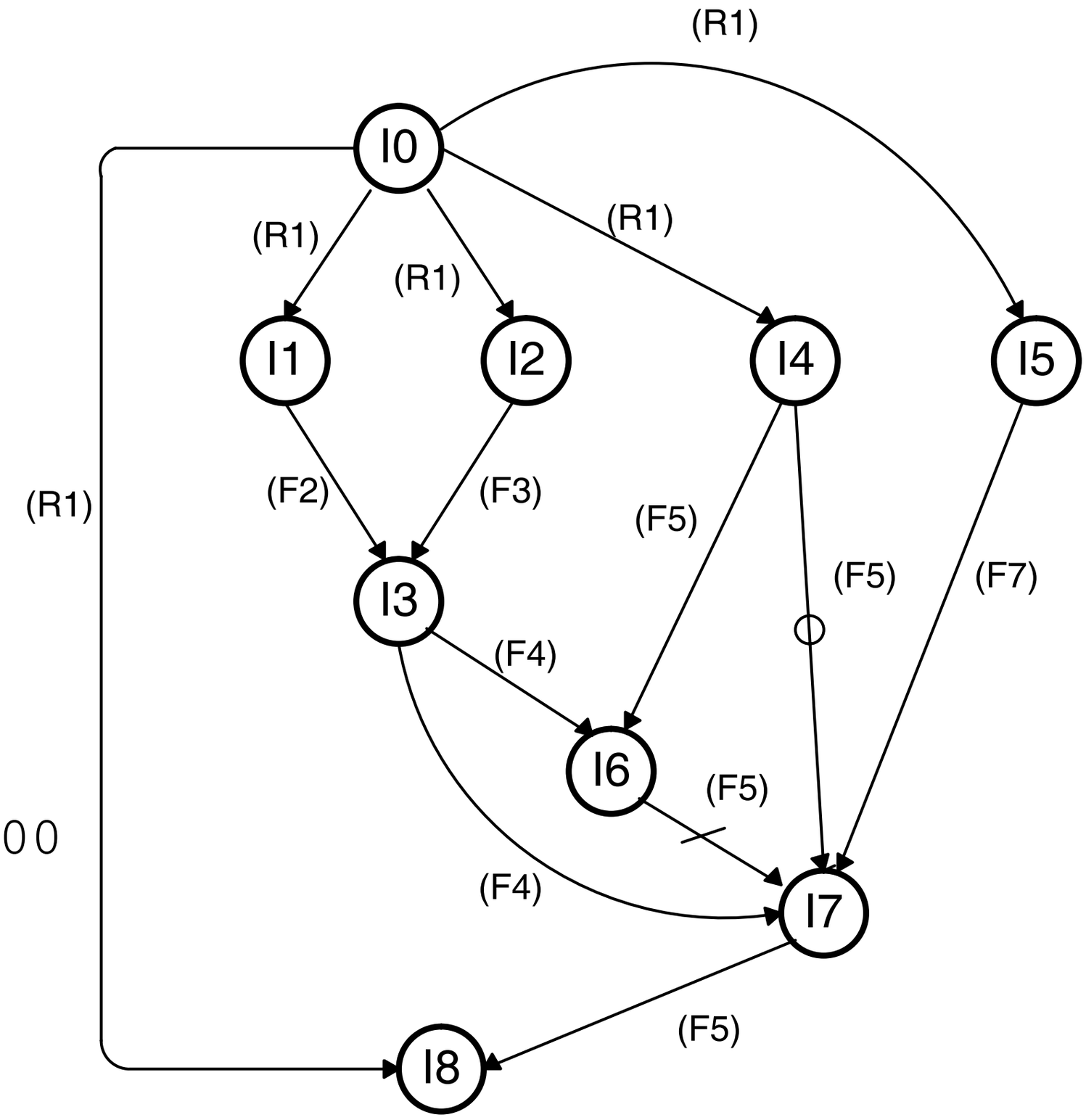
- Data forwarding from VFU takes a single cycle. Result available in dependent VFUs and RF in next cycle.
- After forwarding, source VFU marked as unallocated.

Forwarding to instruction in decode

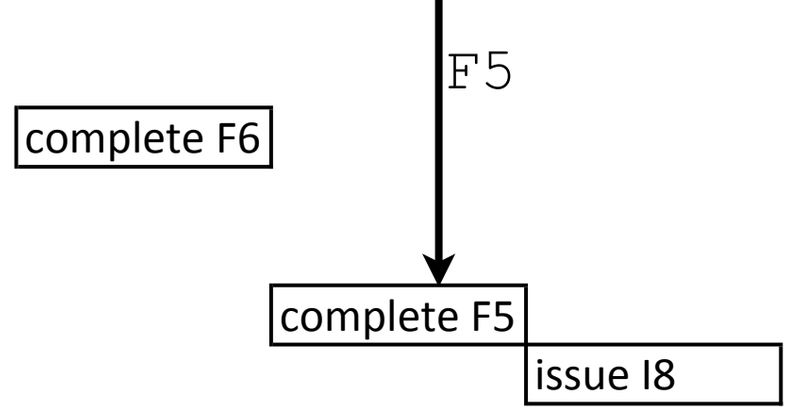
- `I0:  ADD ***R1***,R2,R3` Completing
- `I3:  AND R4,***R1***,R6` Dispatching

- Careful not to miss the result!
- Decode must also monitor CDB

```
(I0)   MOVC R1, #200
(I1)   FLOAD F2, R1, #0
(I2)   FLOAD F3, R1, #4
(I3)   FADD F4, F2, F3
(I4)   FLOAD F5, R1, #8
(I5)   FLOAD F7, R1, #12
(I6)   FSUB F6, F4, F5
(I7)   FMUL F5, F4, F7
(I8)   FSTORE F5, R1, #100
```



cycle	MOVC	FLOAD	FLOAD	FADD	FLOAD	FLOAD	FSUB	FMUL	FSTORE
1	dispatch I0								
2	issue I0	dispatch I1							
3	complete R1		dispatch I2						
4		issue I1	issue I2	dispatch I3					
5					dispatch I4				
6					issue I4	dispatch I5			
7		complete F2	~~*complete F3*~~			issue I5	dispatch I6		
8			complete F3	issue I3				dispatch I7	
9					complete F5				dispatch I8
10						complete F7			
11									
12				complete F4					
13							issue I6	issue I7	
14									
15									
16									
17									
18									
19									
20									
21									




```
(I0)   MOVC R1, #200
(I1)   FLOAD F2, R1, #0
(I2)   FLOAD F3, R1, #4
(I3)   FADD F4, F2, F3
(I4)   FLOAD F5, R1, #8
(I5)   FLOAD F7, R1, #12
(I6)   FSUB F6, F4, F5
(I7)   FMUL F5, F4, F7
(I8)   FSTORE F5, R1, #100
```

Tomasulo's dynamic scheduling algorithm handles *all three types* of data dependencies (flow, anti, output) by simply maintaining *only* the flow dependencies.

(Recall that anti and output dependency ordering constraints have to be maintained only to preserve flow dependencies elsewhere in the code.)

- This is equivalent to getting rid of anti and output dependencies

- Flow dependencies are maintained by copying tags to set up appropriate data flow paths for forwarding:

Side-effects

- With dynamic VFU allocation, Tomasulo's can dynamically unroll loops.
- Tomasulo's allows registers to be updated out-of-order.
  - Presents a challenge for maintaining precise architectural state.

- Some notes about the original implementation of the Toamsulo technique:

- Was deployed only within the floating point part of the 360/91

- Function units were not pipelined

- VFUs implemented on a common physical FU shared a common output buffer that simply held the computed result till it was driven out on the CDB. Till the result was forwarded, the VFU and its associated RSE was not considered free.

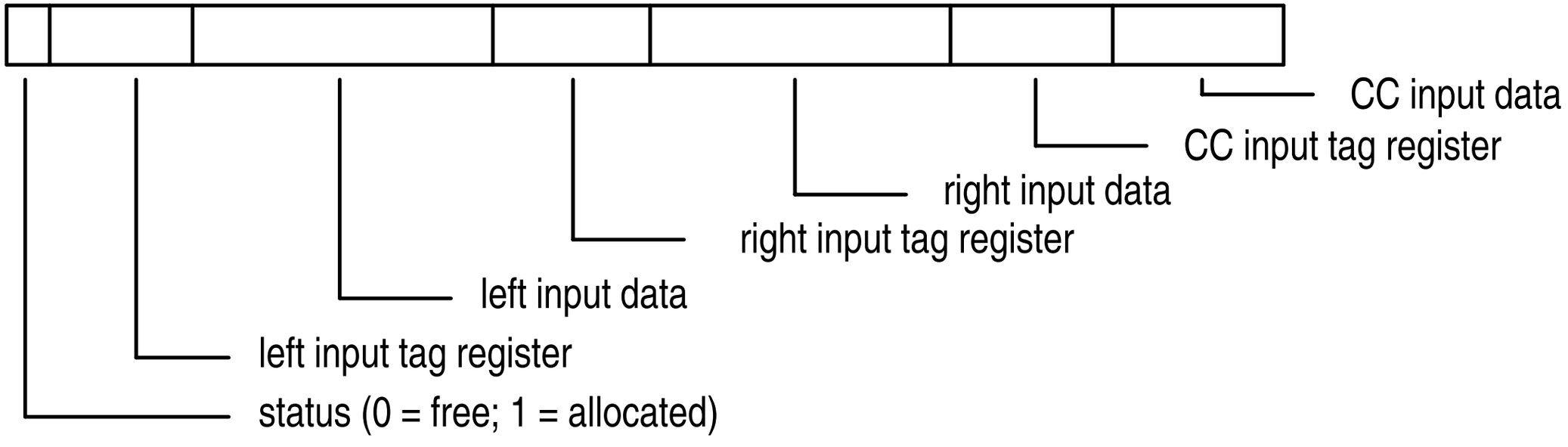
- Precise interrupts (LATER) were not implemented despite out–of–order completions.

- The implementation complexity, was very significant, as discrete components were used in the implementation.

Handling Dependencies over Condition Code in Tomasulo's Technique

- Hardware needed:
- One CC register for integer FUs and another for floating point FUs (ICC and FPCC, respectively).
- Source tag registers associated with these CC registers.
- RSE of VFU extended to have one more slot, to hold the CC flags – this slot has an associated tag field; this extension is needed to handle instructions like ADDC (add with carry) and conditional branches.

VFU entries now look as follows:



- Handling dispatches:

  Instructions dependent on CC flags (ADDC, BC, BZ etc.):
  - handle src registers as usual
  - copy ICC (or FPCC as the case may be) tag and data into the CC input slot in the VFU
  - if instruction can set CC flag, copy id of selected VFU into tag register of this CC (ICC or FCC)

  Other instructions:–
  - handle src registers as usual
  - copy a tag value of 0000 (=valid) into the CC input tag field of the VFU; these instructions do not care about the CC value, so they don't need to wait till the CC becomes valid.
  - if instruction can set CC flag, copy id of selected VFU into tag register of this CC

- Handling startups: execution of an instruction starts up when all the tags associated with all of its inputs are 0000.

- Handling completions:when a VFU completes, it floats out the result, the CC flag values for this result and its id on the CDB. CC values are thus forwarded (like the results) to any waiting instruction.

- Hardware facilities required by Tomasulo's algorithm:

- Common data bus and associated arbitration logic

- Hardware facilities to keep track of instruction and VFU status for dispatching. This information is global in nature.

- Hardware facilities to allow the *physical* FUs to start up (and arbitrate over selecting one of possibly many VFUs that are enabled for startup). This logic is distributed in nature.

- Tag manipulation logic and tag registers

- Associative tag matching logic:

  – Number of comparators needed is D, where D is the number of potential destinations

    D = # of architectural registers + # of RSE slots (all VFU inputs that can hold a register value)

  – If more than one CDB is used to avoid a bottleneck over the use of the CDB, the number of comparators needed increase proportionately.

This is a substantial investment even by todays standard. Most of the performance benefits show up when we have superscalar instruction issue.