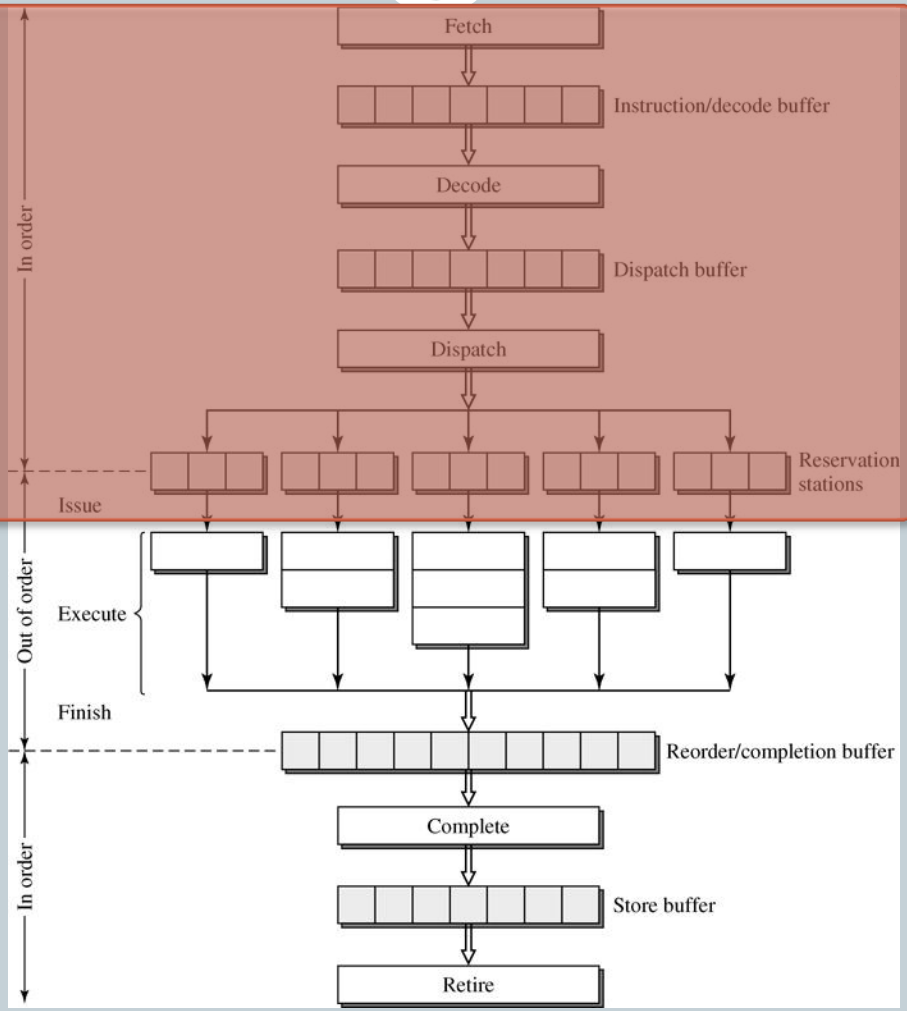
Advanced Computer Architecture

LECTURE 4: DATA STREAMS
INSTRUCTION EXECUTION
INSTRUCTION COMPLETION & RETIREMENT
DATA FLOW & REGISTER RENAMING
DYNAMIC EXECUTION CORE

JAN LEMEIRE

Processor Pipeline

seen so far



Instruction Execution (1)

- Trend towards more and more specialized FUs
- Typically
  - multiple integer FUs
  - multiple floating-point FUs
  - one or more branch units
  - multiple load/store-units
  - SIMD units (SSEx)
  - very recently: cryptographic units

Instruction Execution (2)

- Width determined by fetch, decode and complete width
- Typically #FUs > pipeline width
- Pro
  - Handle dynamic mix of instructions (sometimes not matching mix of FUs), even with increased specialization
- Con
  - Forwarding hardware becomes much more complex
    - order n^2 , slow because of high fan-out (# output lines)
    - need to forward in same cycle as actual computation
    - solution:
      - often don't want need full cross-bar
      - deal with structural hazards instead

Instruction Completion

- After execution
  - instruction leaves FU
  - enters reorder buffer
  - architectural state not yet updated
- Instruction leaves reorder buffer
  - called "completion"
  - in-order
  - architectural state is updated
  - store leaves store queue to enter store buffer

Instruction Retirement

- Stores actually write to memory after completion, during so-called "retirement"
- For other instructions: completion = retirement

Illustration of the issue

- Example code fragment

  - w: R4 = R0 + R8
  - x: R2 = R0 * R4
  - y: R4 = R4 + R8
  - z: R8 = R4 * R2

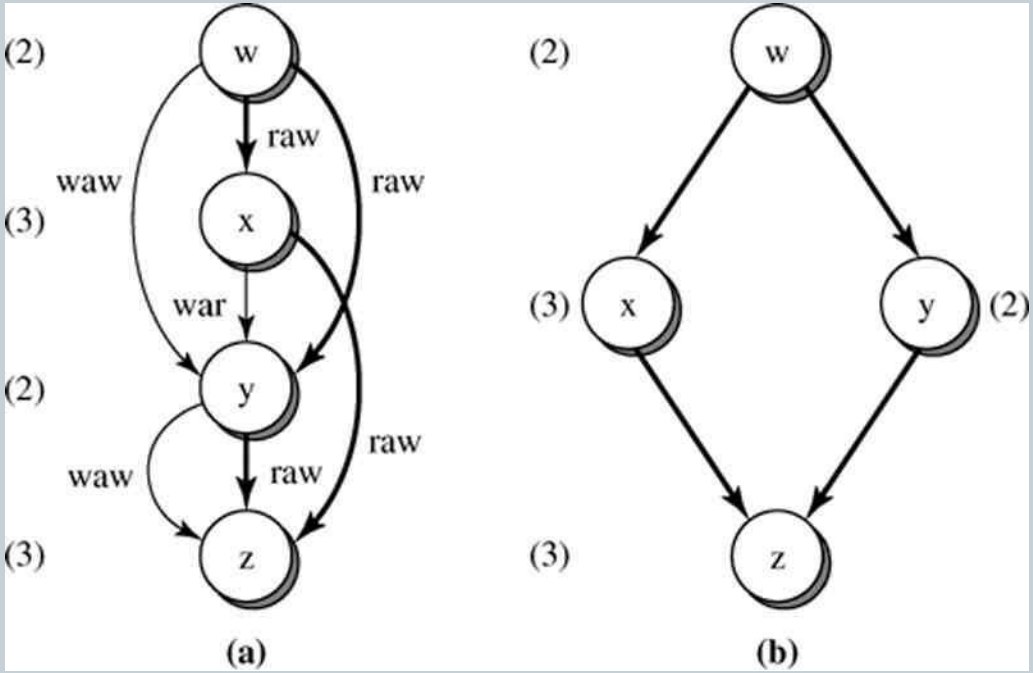
- In how many cycles can this be executed?
  - Addition: 2 cycles
  - Multiplication: 3 cycles

Limits to ILP

- Control flow transfers and pipeline bubbles
- Data hazards
  - RAW: true dependencies
  - WAW: output dependencies (not in in-order processors)
  - WAR: anti dependencies (not in in-order processors)
- Structural hazards
  - resources

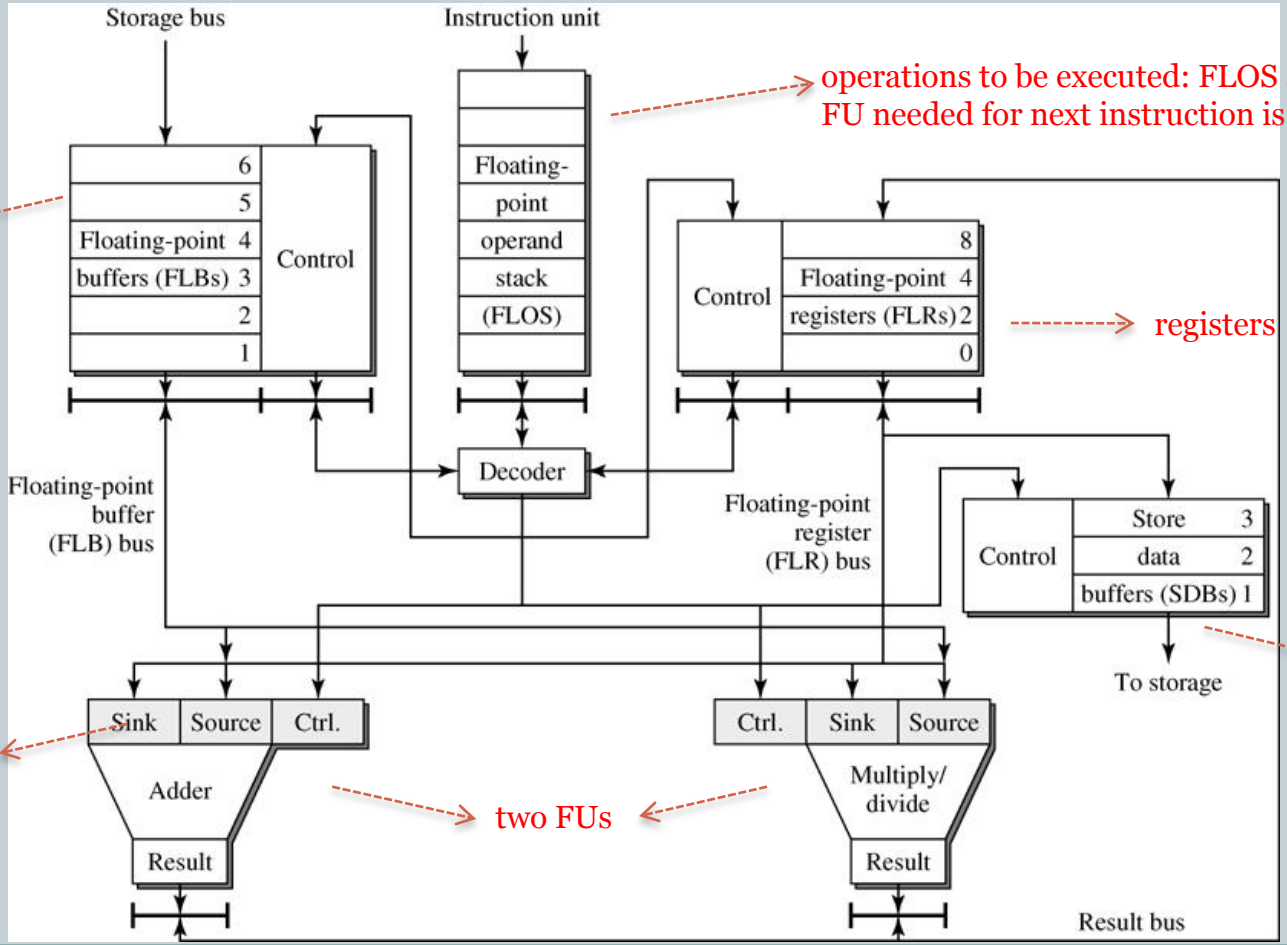
DEPENDENCY GRAPH

- Tomasulo Removes WAW



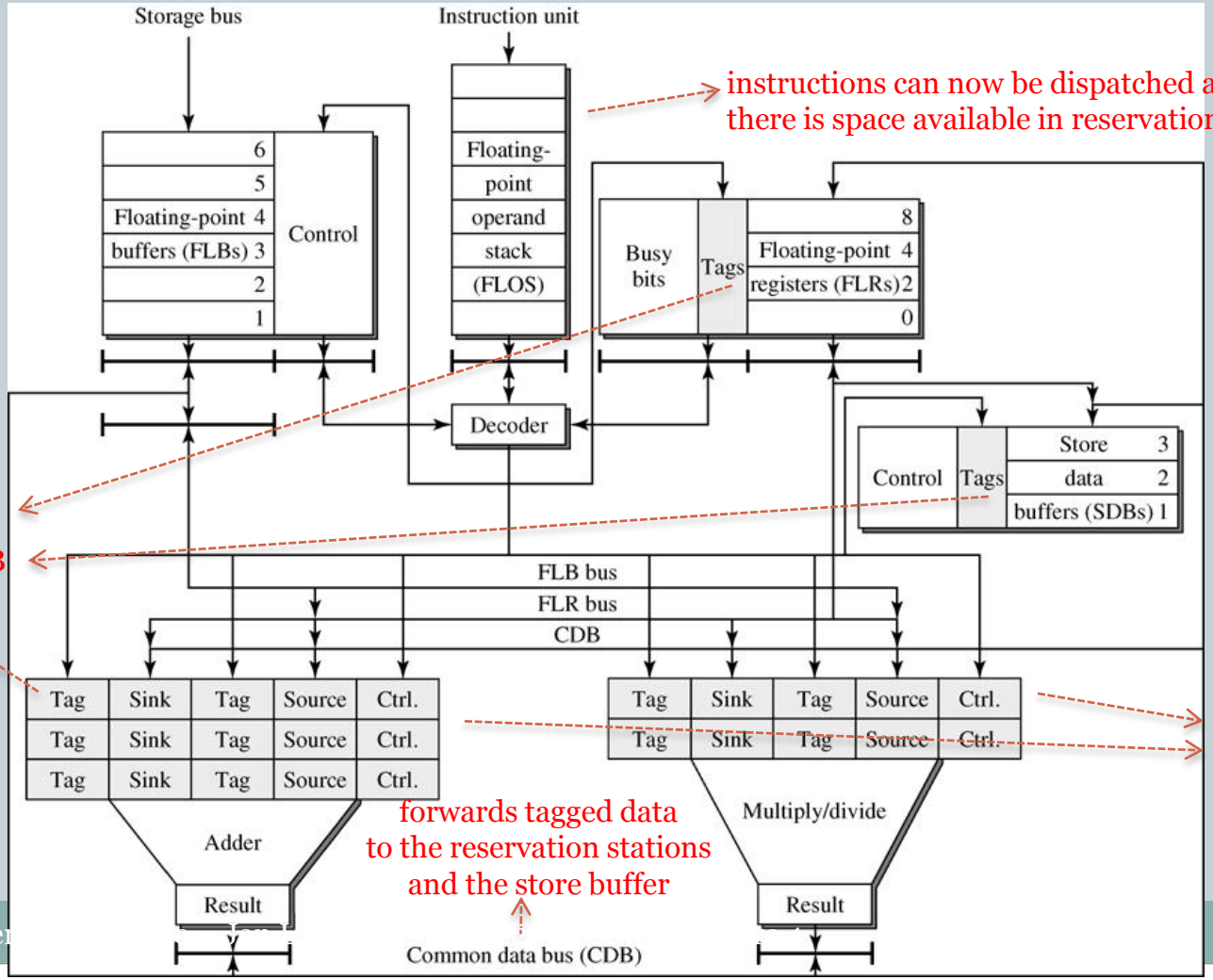
Tomasulo Algorithm (1)

- Original IBM 360 floating-point unit: two-operand register-register micro-ops



Tomasulo Algorithm (2)

- Improved IBM 360/91: added reservation stations, common data bus and tags

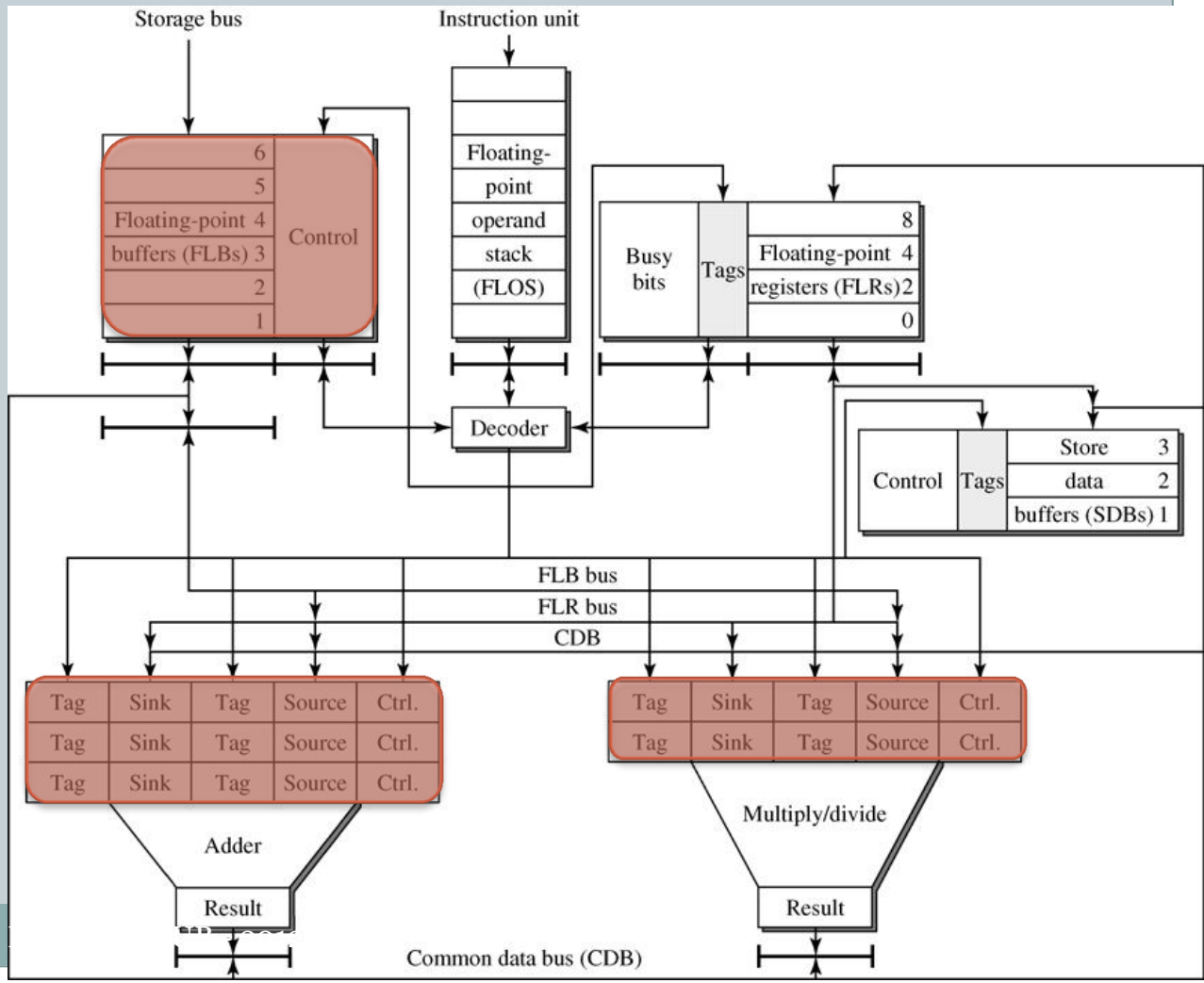


Tomasulo Algorithm (3)

- Improved IBM 360/91: added Reservation Stations (RS), common data bus and tags

Tags need to encode **12 options**:
11 possible sources (6 FLB + 5 RS) for data that is not yet available
+ 1 for when data is present already

=> so 4 bits per tag



Tomasulo Algorithm (5)

CYCLE 1 Dispatched instruction(s): **w, x** (in order)

RS

		Tag	Sink	Tag	Source
w	1	0	6.0	0	7.8
	2				
	3				

w Adder

RS

		Tag	Sink	Tag	Source
x	4	0	6.0	1	----
	5				

Mult/Div

FLR

	Busy	Tag	Data
0			6.0
2	Yes	4	3.5
4	Yes	1	10.0
8			7.8

CYCLE 2 Dispatched instruction(s): **y, z** (in order)

RS

		Tag	Sink	Tag	Source
w	1	0	6.0	0	7.8
y	2	1	----	0	7.8
	3				

w Adder

RS

		Tag	Sink	Tag	Source
x	4	0	6.0	1	----
z	5	2	----	4	----

Mult/Div

FLR

	Busy	Tag	Data
0			6.0
2	Yes	4	3.5
4	Yes	2	10.0
8	Yes	5	7.8

CYCLE 3 Dispatched instruction(s): _____

RS

		Tag	Sink	Tag	Source
	1				
y	2	0	13.8	0	7.8
	3				

y Adder

RS

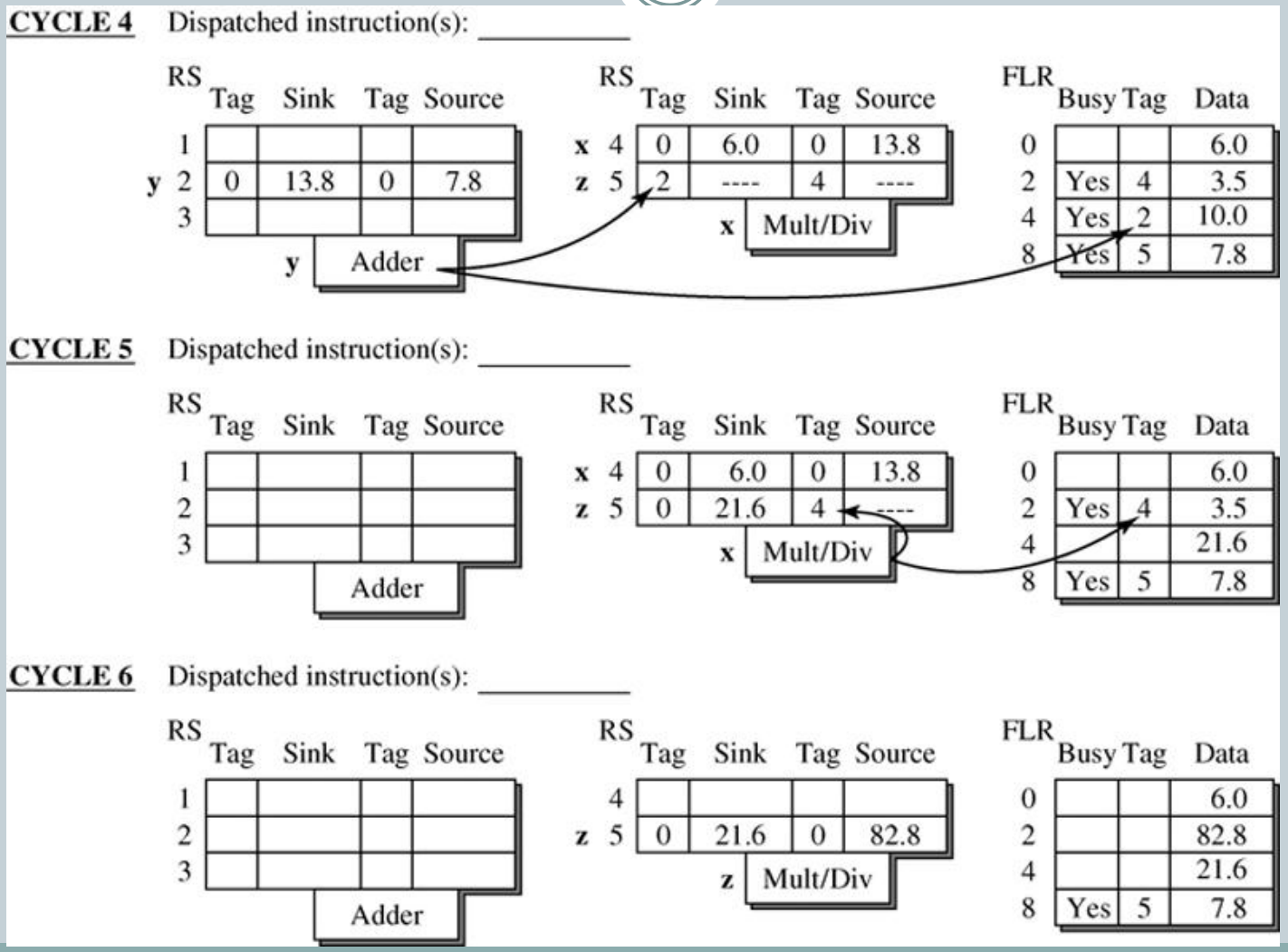
		Tag	Sink	Tag	Source
x	4	0	6.0	0	13.8
z	5	2	----	4	----

x Mult/Div

FLR

	Busy	Tag	Data
0			6.0
2	Yes	4	3.5
4	Yes	2	10.0
8	Yes	5	7.8

Tomasulo Algorithm (6)



Tomasulo Algorithm (7)

1. **Structural (FU) dependence = > virtual FU's**
   - FLOS can hold and decode up to 8 instructions.
   - Instructions are dispatched to the 5 reservation stations (virtual FU's) even though there are only two physical FU's.
   - Hence, structural dependence does not stall dispatching.
2. **True dependence (RAW) = > pseudo operands + result forwarding**
   - If an operand is available in FLR, it is copied to a reservation station entry.
   - If an operand is not available (i.e. there is pending write), then a tag is copied to the reservation station entry instead. This tag identifies the source of the pending write. This instruction then waits in its reservation station for the true dependence to be resolved.
   - When the operand is finally produced by the source (ID of source = tag value), this source unit asserts its ID, i.e. its tag value, on the CDB followed by broadcasting of the operand on the CDB.
   - All the reservation station entries and the FLR entries and SDB entries carrying this tag value in their tag fields will detect a match of tag values and latch in the broadcasted operand from the CDB.
   - Hence, true dependence does not block subsequent independent instructions and does not stall a physical FU. Forwarding also minimizes delay due to true dependence.

Tomasulo Algorithm (8)

3. **Anti-dependence (WAR)** = > operand copying

   - If an operand is available in FLR, it is copied to a reservation station entry.
   - By copying this operand to the reservation station, all anti-dependences due to future writes to this same register are resolved.
   - Hence, the reading of an operand is not delayed, possibly due to other dependences, and subsequent writes are also not delayed.

Tomasulo Algorithm (9)

4. **Output dependence (WAW)** = > register renaming + result forwarding
   - If a register is waiting for a pending write, its tag field will contain the ID, or tag value, of the source for that pending write.
   - When that source eventually produces the result, that result will be written into the register via the CDB.
   - It is possible that prior to the completion of the pending write, another instruction can come along and also has that same register as its destination register.
   - If this occurs, the operands (or pseudo operands) needed by this instruction are still copied to an available reservation station. In addition, the tag field of the destination register of this instruction is updated with the ID of this new reservation station, i.e. the old tag value is overwritten. This will ensure that the said register will get the latest value, i.e. the late completing earlier write cannot overwrite a later write.
   - However, the forwarding to instructions between those two output dependent instructions will still use the old tag.
   - Hence, the output dependence is resolved without stalling a physical functional unit, not requiring additional buffers to ensure sequential write back to the register file.

Tomasulo Algorithm (10)

- Supports out of order execution of instructions.
- Resolves dependences dynamically using hardware.
- Attempts to delay the resolution of dependencies as late as possible.
- Structural dependence does not stall issuing; virtual FU's in the form of reservation stations are used.
- Output dependence does not stall issuing; copying of old tag to reservation station and updating of tag field of the register with pending write with the new tag.
- True dependence with a pending write operand does not stall the reading of operands; pseudo operand (tag) is copied to reservation station.
- Anti-dependence does not stall write back; earlier copying of operand awaiting read to the reservation station.
- Can support sequence of multiple output dependences.
- Forwarding from FU's to reservation stations bypasses the register file.

Tomasulo Algorithm (11)

	IBM 360/91	Modern processors
Width	Peak IPC = 1	4+
Structural hazards	2 FPU Single CDB	Many FU Many busses
Anti-dependences	Operand copy	Reg. renaming
Output dependences	Renamed reg. tag	Reg. renaming
True dependences	Tag-based forw.	Tag-based forw.
Exceptions	Imprecise	Precise (ROB)
Implementation	3 x 66” x 15” x 78” 60ns cycle time 11-12 gate delays per pipe stage >$1 million	1 chip 300ps < $100

Data Flow ILP Limit (1) [244]

- Suppose we have a program

```
w[i+k].ip = z[i].rp + z[m+i].rp;
w[i+j].rp = e[k+1].rp * (z[i].rp - z[m+i].rp) - e[k+1].ip * (z[i].ip - z[m+i].ip);
```

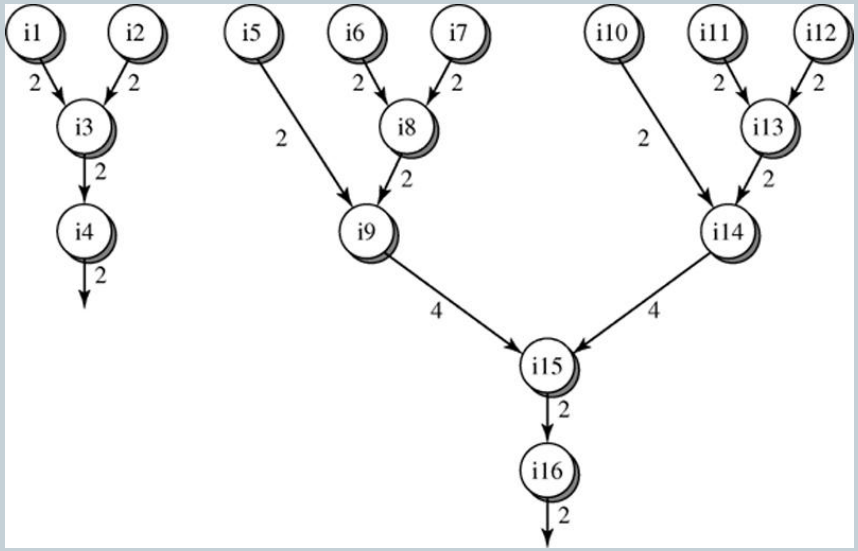
(a)

```
i1:  f2 ← load,4(r2)
i2:  f0 ← load,4(r5)
i3:  f0 ← fadd,f2,f0
i4:  4(r6) ← store,f0
i5:  f14 ← laod,8(r7)
i6:  f6 ← load,0(r2)
i7:  f5 ← load,0(r3)
i8:  f5 ← fsub,f6,f5
i9:  f4 ← fmul,f14,f5
i10:  f15 ← load,12(r7)
i11:  f7 ← load,4(r2)
i12:  f8 ← load,4(r3)
i13:  f8 ← fsub,f7,f8
i14:  f8 ← fmul,f15,f8
i15:  f8 ← fsub,f4,f8
i16:  0(r8) ← store,f8
```

(b)

Data Flow ILP Limit (2) [245]

- Depth of data dependence graph determines minimal number of cycles on any machine
  - only RAW dependencies
- Further limits imposed by finite number of resources
  - nr of FUs
  - pipeline width
  - nr of registers
- Registers are reused in code in flight
  - because of register allocation
  - because of small loops
  - reuse poses additional limits because of WAW and WAR dependencies



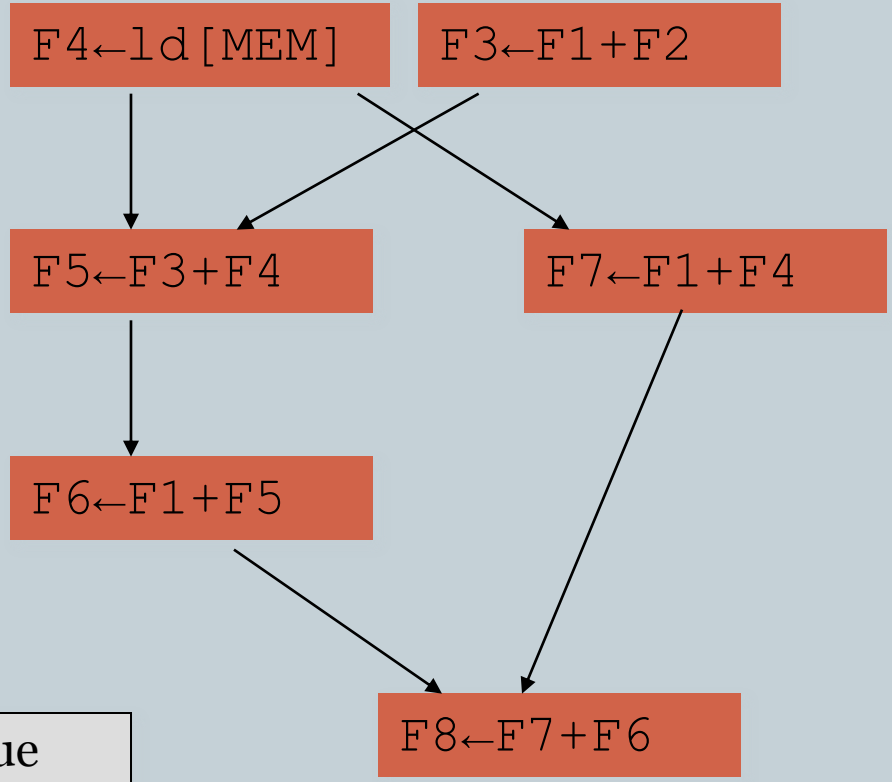
Register Renaming [237]

- Reuse means storing more than one value in a register
  - until older value is needed, it cannot be overwritten by newer value
- Central idea:
  - do not limit storage of operands to architectural registers
  - add other "renaming registers"
  - this provides storage space to avoid reuse
  - rename *architectural registers* (visible to programmer) in operands to avoid reuse in instruction window
    - removes WAR dependencies
    - removes WAW dependencies
  - Idea: write to a register no more than once
  - all in hardware

Register Renaming: example

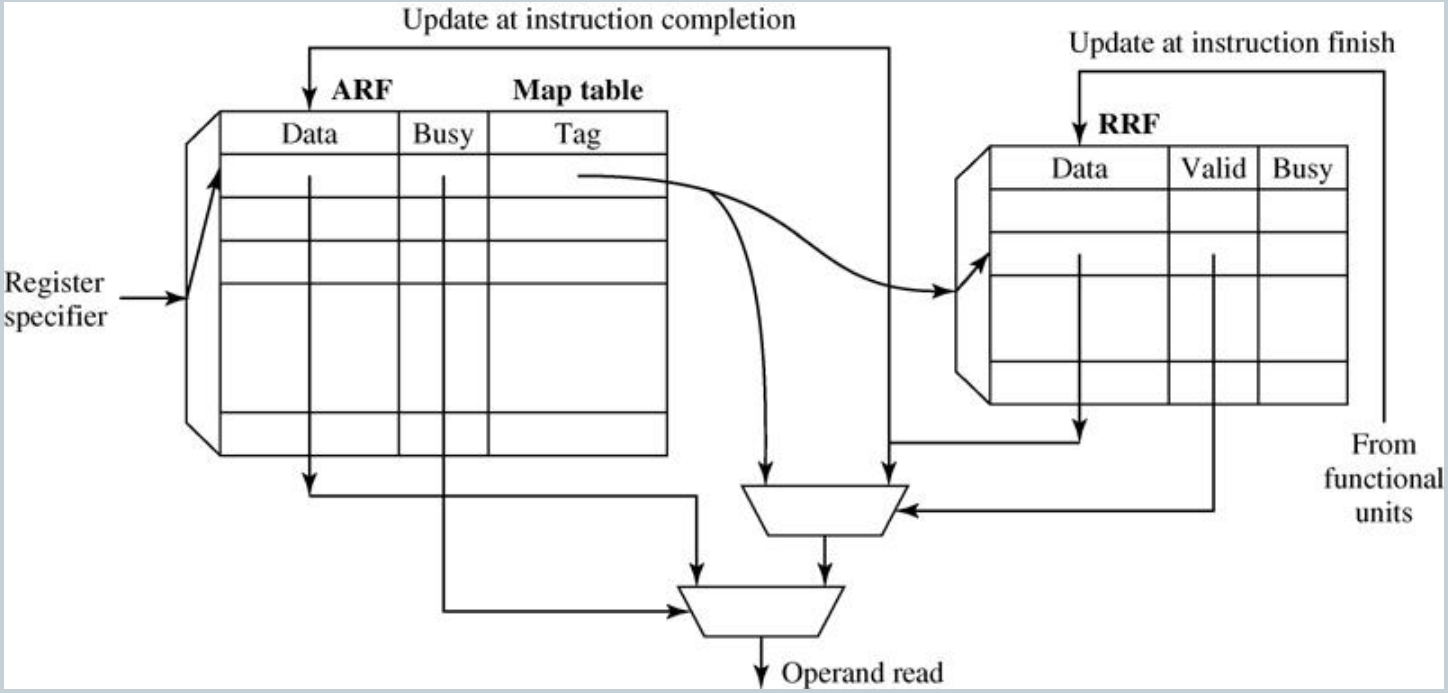
R4←ld[MEM]	F4←ld[MEM]
R3←R1+R2	F3←F1+F2
R5←R3+R4	F5←F3+F4
R6←R1+R5	F6←F1+F5
R3←R1+R4	F7←F1+F4
R5←R3+R6	F8←F7+F6

every instruction in flight gets a unique destination operand (single assignment)



Register Renaming: tasks

- Three tasks per instruction



1) source read 2) destination allocate 3) register update

1) and 2) in pipeline front-end, 3) in pipeline back-end

Register Renaming: (1) Source read

- Busy bit in ARF = 0
  - No instruction in flight that will write to this register
  - Get operand from this ARF register
- Busy bit in ARF = 1
  - Instruction in flight that will write to this ARF registers
  - Tag in map table identifies corresponding RRF register
  - 2 possibilities
    - Valid bit in RRF = 1
      - Instruction is in flight but not yet completed, so read the operand from RRF
    - Valid bit in RRF = 0
      - Instruction that will write is not yet executed, feed tag of corresponding RRF to reservation buffer (where we will wait for the operand to arrive)

Register Renaming: (2) Destination allocate

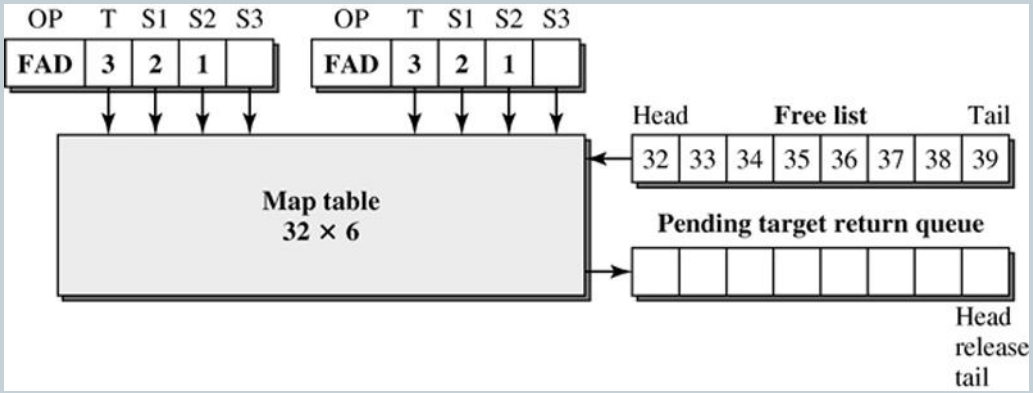
- Index ARF with register number
- Set busy bit in ARF to 1
- Find free register (busy bit = 0) in RRF
- Set busy bit in RRF to 1
- Set valid bit in RRF to 0
- Store rename register tag in map table of ARF register

Register Renaming: (3) Register Update

- When instruction leaves FU (finishes)
  - Write result value in RRF at correct place
  - Set valid bit in RRF to 1
- When instruction leaves the reorder *(completes)*
  - Copy the result value from RRF to ARF
  - Set busy bit in RRF to 0
  - This happens in-order!
- Both steps can be in consecutive cycles or with many cycles in between …

Register Renaming (7)

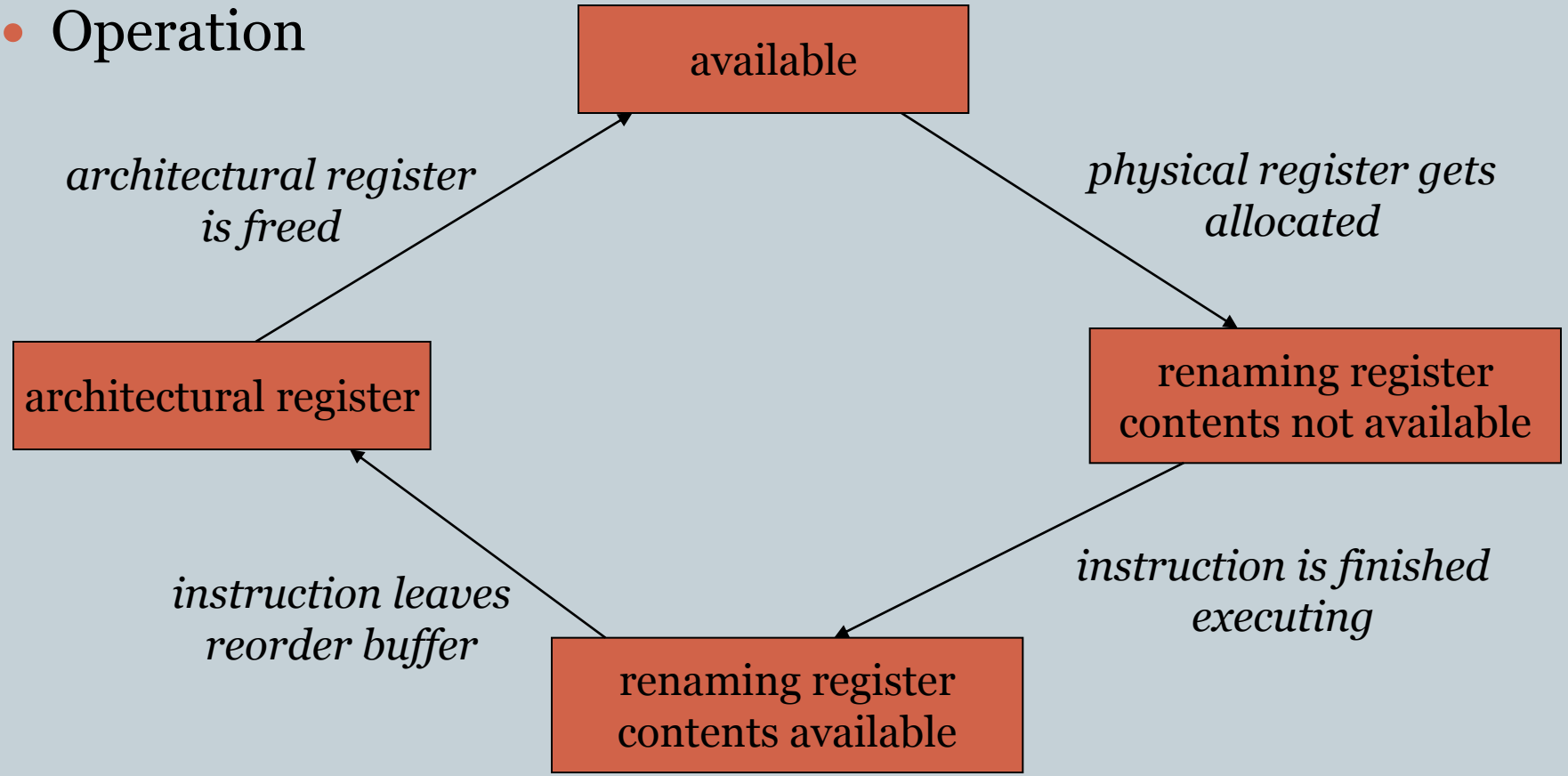
- Alternative: one combined register file (pooled or merged)
  - architectural registers are part of one larger register file
  - map table remembers which of the registers are considered as architectural ones (this may change)



  - Con: before context switch, need to know which architectural registers to save
  - Pro: no copying of data between RRF and ARF

Register Renaming (8)

- Operation



Register Renaming (9)

- Initialisation
  - All physical registers that are mappings of architectural registers are in the "architectural register" state
  - The other physical registers are "available"
- for the input operand
  - Read physical register corresponding to the architectural register from the mapping table
- for the output operand
  - Select an "available" physical register, and change its condition to "renaming register, content not available" update the mapping table
  - If the original physical register was in the "architectural register" state, its new state becomes "available"
  - If there is no "available" register, block until some become available

Register Renaming (10)

- When an instruction finishes execution
  - Change the state to "renaming register, content available"
- When an instruction leaves the reorder buffer (completion)
  - change the state into "architectural register",
  - the previous physical register associated with this architectural register becomes "available"
    - Its previous value will not be necessary anymore in case of exceptions
    - Its previous mapping was remembered in the reorder buffer

Register Renaming (11)


```
R4←ld[MEM]
R3←R1+R2
R5←R3+R4
R5←R1+R5
R3←R3+R5
R5←R3+R5
```

```
R1 : F1
R2 : F2
R3 : F3
R4 : F4
R5 : F5
```

```
F1 (R1): AR
F2 (R2): AR
F3 (R3): AR
F4 (R4): AR
F5 (R5): AR
F6 (--): AV
F7 (--): AV
```

AR = architectural register
AV = available
RR = renaming register

Register Renaming (12)


```
R4←ld[MEM]
F6←ld[MEM]
R3←R1+R2
R5←R3+R4
R5←R1+R5
R3←R3+R5
R5←R3+R5
```

```
R1 : F1
R2 : F2
R3 : F3
R4 : F6
R5 : F5
```

```
F1 (R1): AR
F2 (R2): AR
F3 (R3): AR
F4 (--): AV
F5 (R5): AR
F6 (R4): RR
F7 (--): AV
```

AR = architectural register
AV = available
RR = renaming register

Register Renaming (13)

R4←ld[MEM]	F6←ld[MEM]
R3←R1+R2	F7←F1+F2
R5←R3+R4	
R5←R1+R5	
R3←R3+R5	
R5←R3+R5	

```
R1 : F1
R2 : F2
R3 : F7
R4 : F6
R5 : F5
```

```
F1 (R1): AR
F2 (R2): AR
F3 (--): AV
F4 (--): AV
F5 (R5): AR
F6 (R4): RR
F7 (R3): RR
```

AR = architectural register
AV = available
RR = renaming register

Register Renaming (14)

R4←ld[MEM]	F6←ld[MEM]
R3←R1+R2	F7←F1+F2
R5←R3+R4	F3←F7+F6
R5←R1+R5	
R3←R3+R5	
R5←R3+R5	

```
R1 : F1
R2 : F2
R3 : F7
R4 : F6
R5 : F3
```

```
F1 (R1): AR
F2 (R2): AR
F3 (R5): RR
F4 (--): AV
F5 (--): AV
F6 (R4): RR
F7 (R3): RR
```

AR = architectural register
AV = available
RR = renaming register

Register Renaming (15)

R4←ld[MEM]	F6←ld[MEM]
R3←R1+R2	F7←F1+F2
R5←R3+R4	F3←F7+F6
R5←R1+R5	F4←F1+F3
R3←R3+R5	
R5←R3+R5	

```
R1 : F1
R2 : F2
R3 : F7
R4 : F6
R5 : F4
```

```
F1 (R1): AR
F2 (R2): AR
F3 (R5): RR
F4 (R5): RR
F5 (--): AV
F6 (R4): RR
F7 (R3): RR
```

AR = architectural register
AV = available
RR = renaming register

Register Renaming (16)

R4←ld[MEM]	F6←ld[MEM]
R3←R1+R2	F7←F1+F2
R5←R3+R4	F3←F7+F6
R5←R1+R5	F4←F1+F3
R3←R3+R5	F5←F7+F4
R5←R3+R5	

```
R1 : F1
R2 : F2
R3 : F5
R4 : F6
R5 : F4
```

```
F1 (R1): AR
F2 (R2): AR
F3 (R5): RR
F4 (R5): RR
F5 (R3): RR
F6 (R4): RR
F7 (R3): RR
```

AR = architectural register
AV = available
RR = renaming register

Register Renaming (17)

R4←ld[MEM]	F6←ld[MEM]
R3←R1+R2	F7←F1+F2
R5←R3+R4	F3←F7+F6
R5←R1+R5	F4←F1+F3
R3←R3+R5	F5←F7+F4
R5←R3+R5	

```
R1 : F1
R2 : F2
R3 : F5
R4 : F6
R5 : F4
```

```
F1 (R1): AR
F2 (R2): AR
F3 (R5): RR
F4 (R5): RR
F5 (R3): RR
F6 (R4): RR
F7 (R3): RR
```

Register renaming now blocks because there are no more available registers.

Register Renaming (18)


```
R4←ld[MEM]
R3←R1+R2
R5←R3+R4
R5←R1+R5
R3←R3+R5
R5←R3+R5
```

```
F3←F7+F6
F4←F1+F3
F5←F7+F4
```

```
R1 : F1
R2 : F2
R3 : F5
R4 : F6
R5 : F4
```

```
F1 (R1): AR
F2 (R2): AR
F3 (R5): RR
F4 (R5): RR
F5 (R3): RR
F6 (R4): AR
F7 (R3): AR
```

Now suppose the first two instructions leave the reorder buffer

Register Renaming (19)

R4←ld[MEM]

R3←R1+R2

R5←R3+R4

R5←R1+R5

R3←R3+R5

R5←R3+R5

F5←F7+F4

```
R1 : F1
R2 : F2
R3 : F5
R4 : F6
R5 : F4
```

```
F1 (R1): AR
F2 (R2): AR
F3 (--): AV
F4 (R5): AR
F5 (R3): RR
F6 (R4): AR
F7 (R3): AR
```

And the next two instructions leave the reorder buffer as well …

Register Renaming (20)


```
R4←ld[MEM]
R3←R1+R2
R5←R3+R4
R5←R1+R5
R3←R3+R5
R5←R3+R5
```

```
F5←F7+F4
F3←F5+F4
```

```
R1 : F1
R2 : F2
R3 : F5
R4 : F6
R5 : F3
```

```
F1 (R1): AR
F2 (R2): AR
F3 (R5): RR
F4 (R5): AR
F5 (R3): RR
F6 (R4): AR
F7 (R3): AR
```

Register Renaming (21)


```
R4←ld[MEM]
R3←R1+R2
R5←R3+R4
R5←R1+R5
R3←R3+R5
R5←R3+R5
```

```
R1 : F1
R2 : F2
R3 : F5
R4 : F6
R5 : F3
```

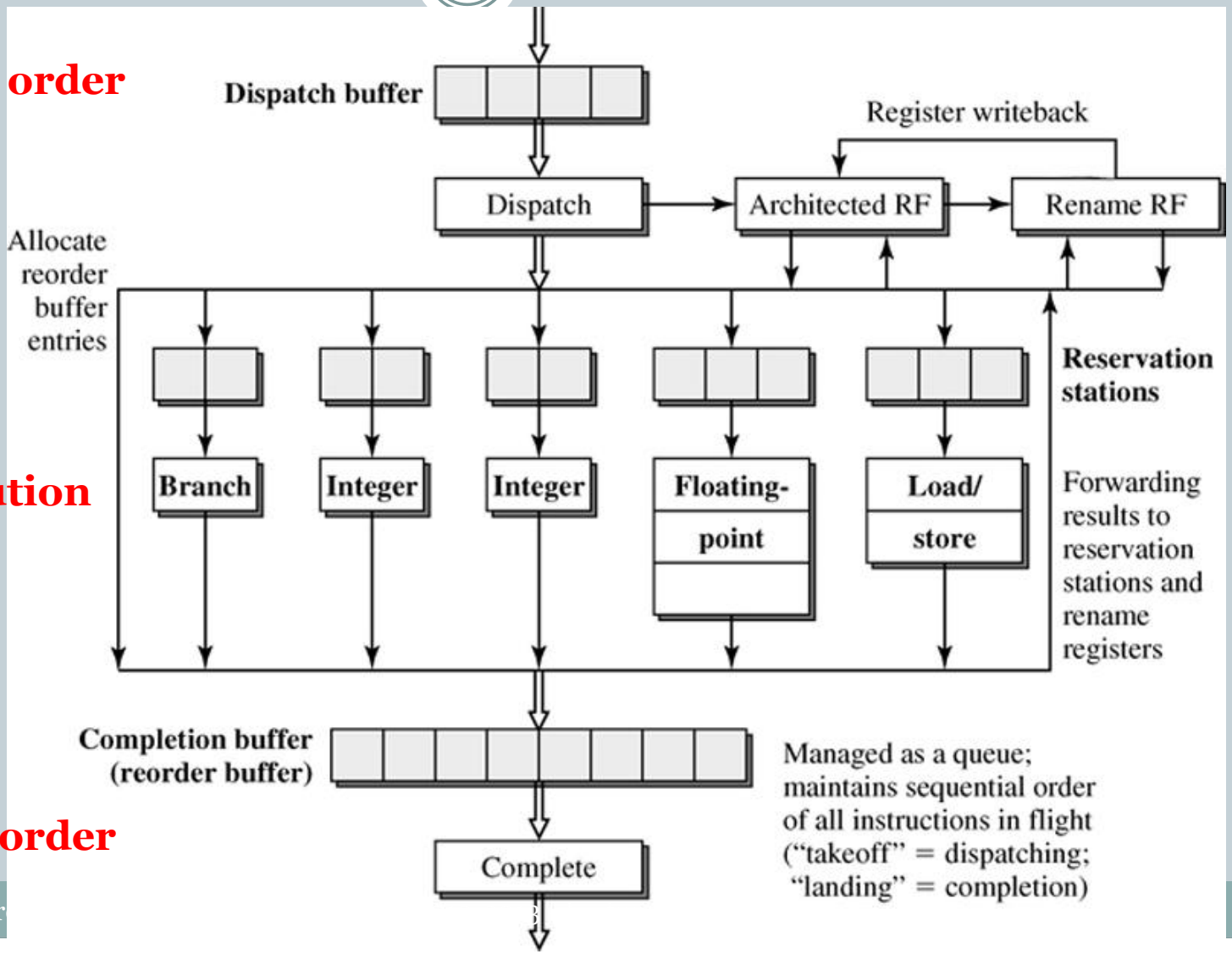
```
F1 (R1): AR
F2 (R2): AR
F3 (R5): AR
F4 (--): AV
F5 (R3): AR
F6 (R4): AR
F7 (--): AV
```

Dynamic Execution Core [254]

front-end is in order

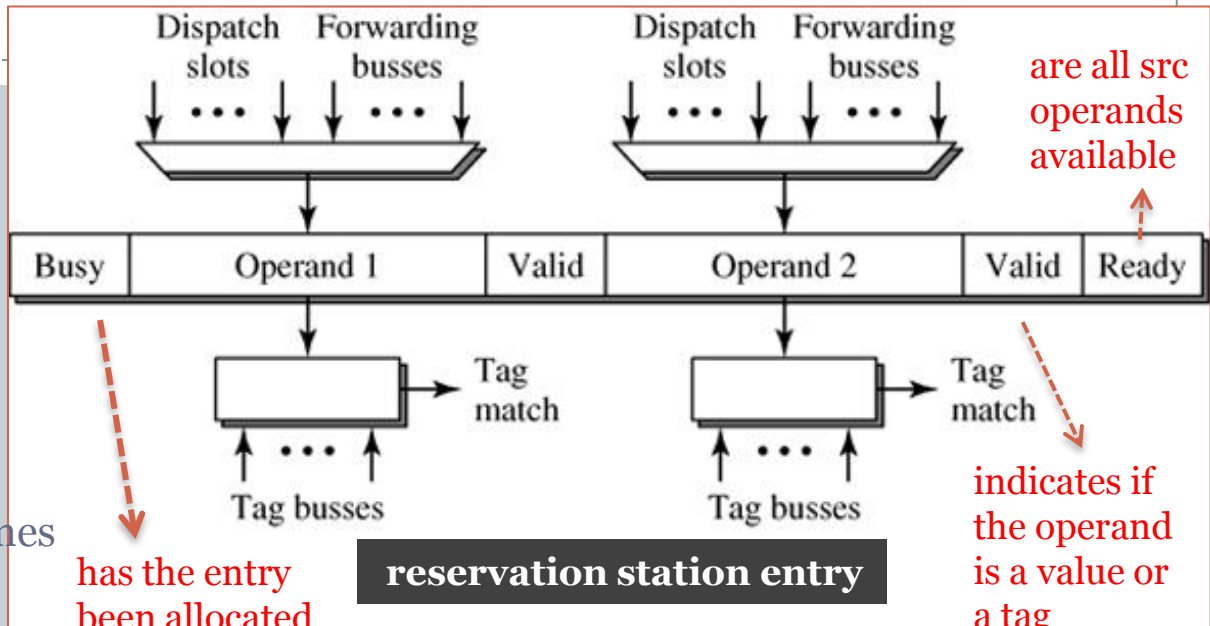
in between: data-flow-like dynamic execution core

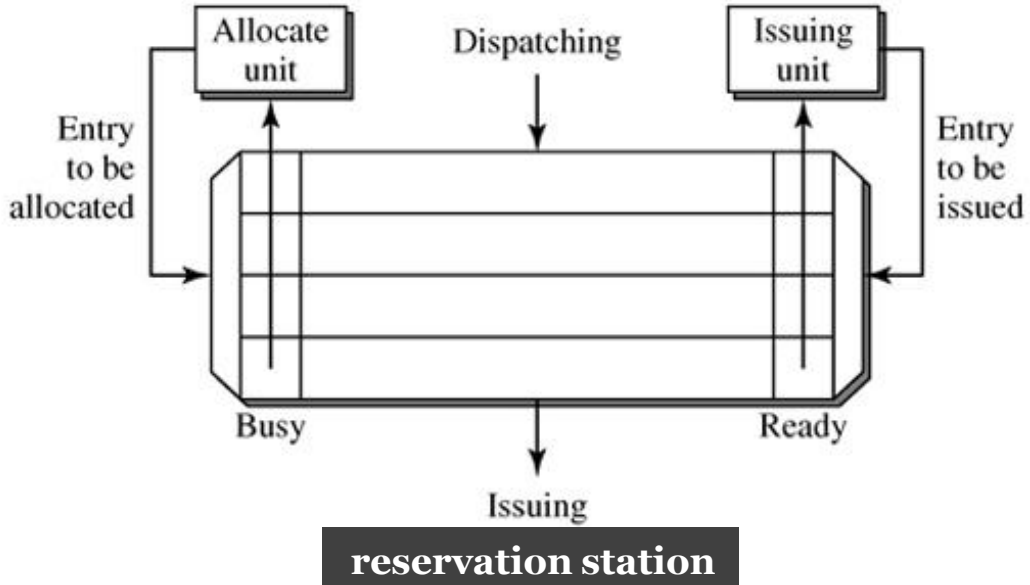
back-end is in order



Reservation Stations (1) [256]

1. dispatching
   - search non-busy entry
   - load operands and tags into entry
   - set busy bit
2. waiting
   - monitor the tag bus
   - latch operand when it becomes available
   - set ready bit if all operands are valid (= "wake-up")
3. issuing
   - "select" a ready instruction
   - reset busy bit
   - with scheduling heuristic
   - complexity increases with #inputs and #outputs/cycle



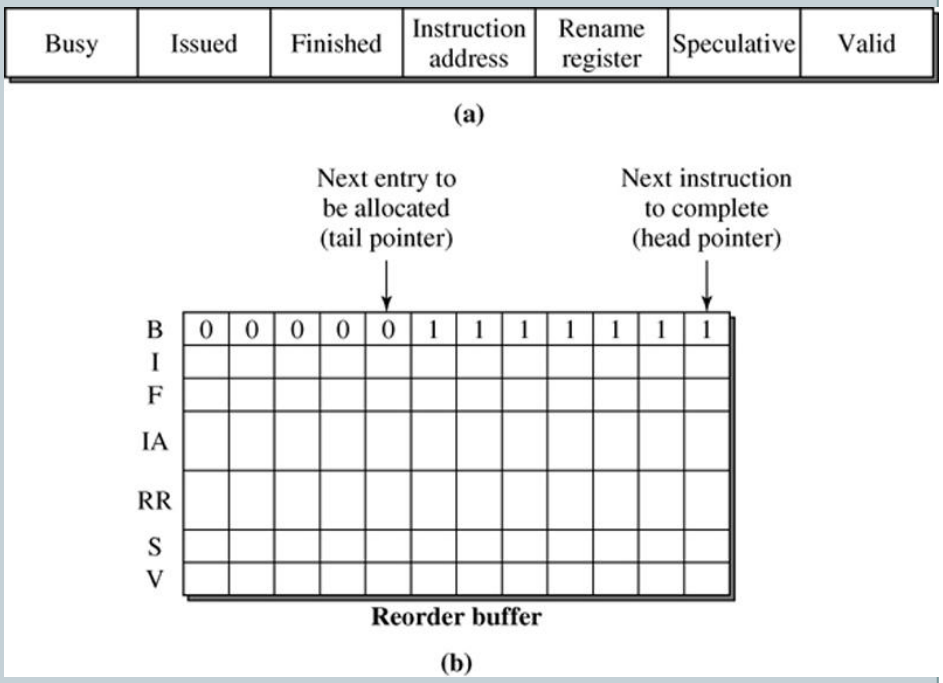


Reservation Stations (2)

- Long latency operations can be reselected and reissued
  - for example loads that miss in the cache
  - instead of blocking the pipeline, go back to waiting state
  - reissue the instruction later when data is available

Reorder Buffer

- Tracks the state of all instructions in flight
  - being executed
  - finished execution but not completed
  - awaiting execution in reservation slot
- Bits indicate state
  - possibly tags when speculating beyond multiple branches
- Implemented as a circular queue
- Completion bandwidth
  - determins number of instructions that can complete from head pointer
  - depends on logic and the number of register writeback ports



Instruction Window

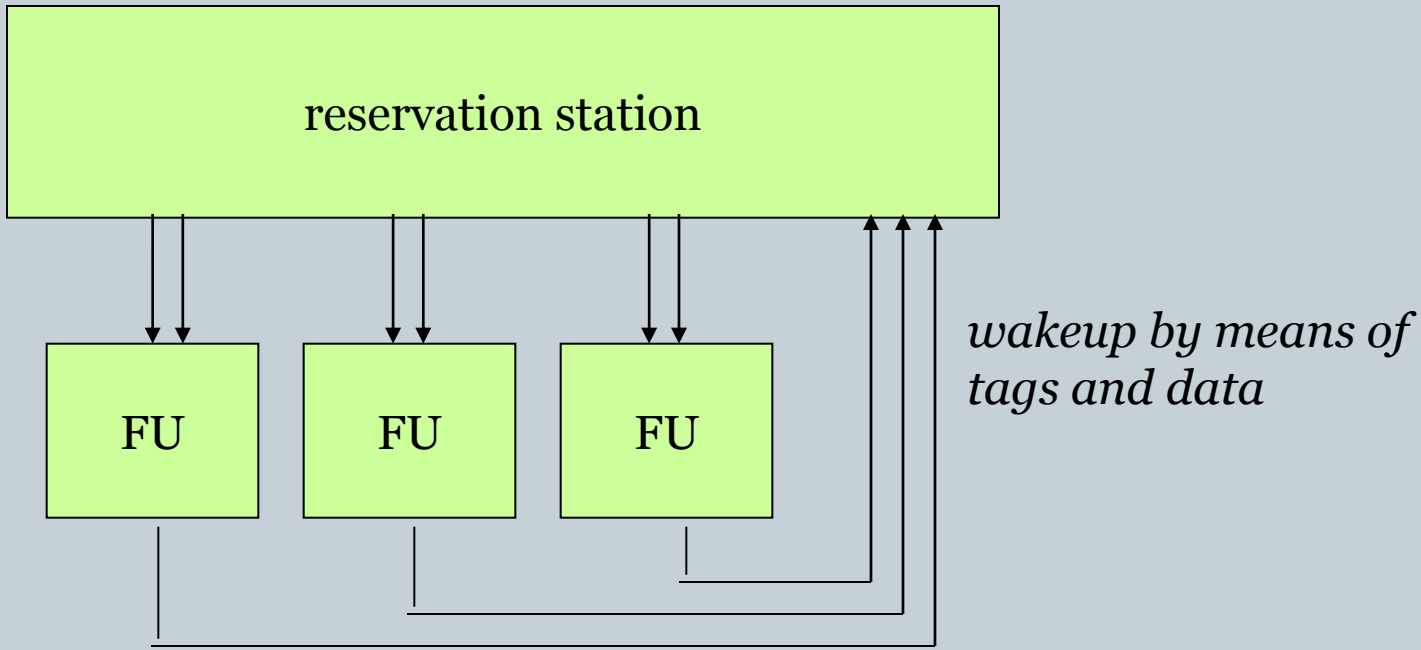
- Reservation station and reorder buffer can be combined in one structure:
  the instruction window
- Pro
  - no duplication of data (i.e. total area is smaller)
  - everything combined in one controller
- Con
  - one large structure instead of two smaller ones
  - hence long connections
  - is problematic for high frequencies

Dynamic Instruction Scheduling (1)

- Two variations on reservation station/instruction window
  - data captured scheduling
    - entries in reservation station hold values of source operands
    - contents read from RF before dispatch
    - see previous slides
  - non-data captured scheduling
    - entries in reservation station do not hold values of source operands
    - values are read from RF after selection, just prior to execution
    - there is no OF stage in pipeline front-end
    - only renaming registers occur in reservation stations
    - values are fetched from renaming registes on wake-up
    - is smaller and less complex
      - no register contents
      - no wires to distribute values over all instructions in reservation stations or in instruction window

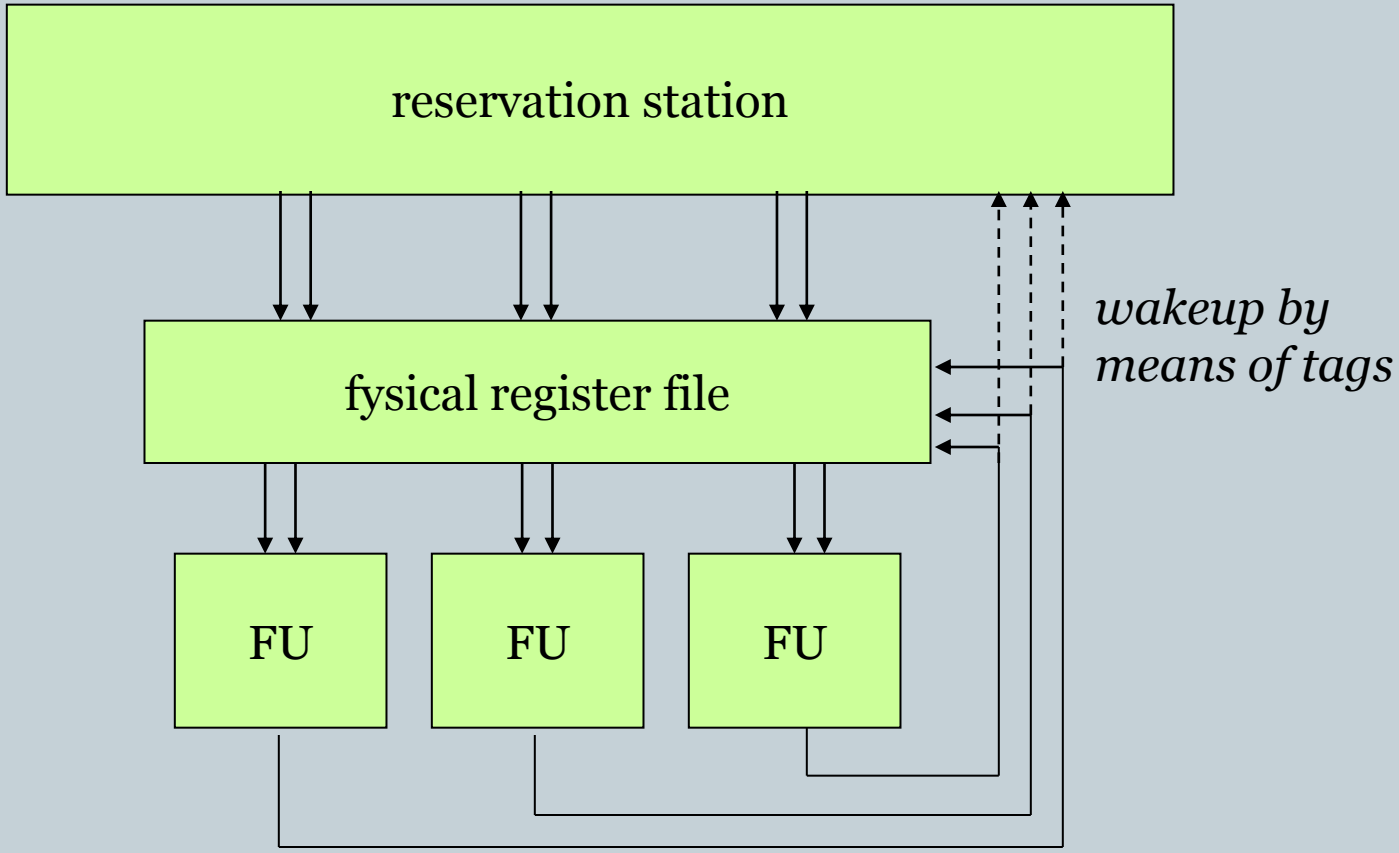
Dynamic Instruction Scheduling (2)

- Data captured



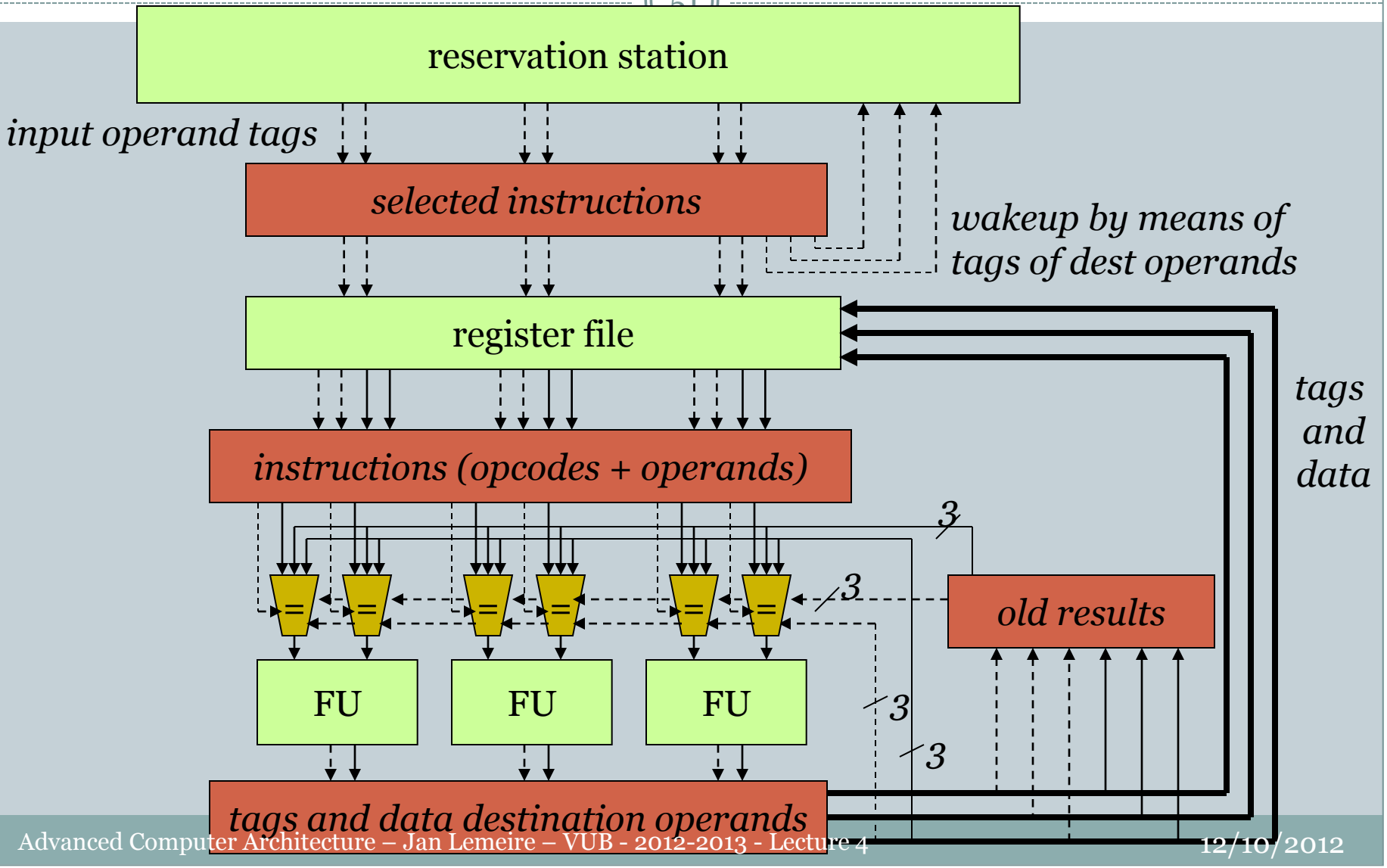
Dynamic Instruction Scheduling (3)

- Non Data captured



if not pipelined: wake-up, selection, reading the register file, execution and tag distribution must happen all in the same clock cycle

Dynamic Instruction Scheduling (4)



Dynamic Instruction Scheduling (5)

- Four pipeline stages
  - Wake-up and select
  - Read register file
  - Execute instruction
  - Write result in register file
- Consequences
  - Wake-up and selection in one cycle
  - Instruction execution and bypassing in one cycle
  - Hence no increased instruction execution latency

So far: reach the data flow limit

- Try all kinds of techniques
  - branch prediction
  - efficient fetching
  - efficient decoding (caching decoded info)
  - dynamic execution
  - register renaming

- Is this it?
- Can't we do better?

Breaking the data flow limit (1)

- Value prediction
  - like branch prediction, predict operand value at beginning of pipeline
  - don't need to wait for operands
  - based on value locality
    - some instructions always produce the same value
    - others generate sequences of values with patterns (loop indexes)
  - store past in a table, and predict future
  - need to validate prediction
  - misprediction should have minimal cost
  - hybrid designs obtain 8-23% IPC increase on SPEC benchmarks

Breaking the data flow limit (2)

- Dynamic Instruction Reuse
  - often instructions are executed repeatedly on the same inputs
  - don't need to to reexecute them if outcome was saved somewhere
  - even happens for sequences of instructions
  - dynamic instruction reuse
    - eliminates nodes (computations) and edges in the data flow graph
    - whereas value prediction only eliminated edges
  - is not speculative, so no validation required

Breaking the data flow limit (3)

- Micro-instruction fusion
  - CISC ISA instructions are broken down into micro-instructions
  - ISA remains unchanged
  - micro-architecture changes over time (pipeline depth, etc.)
  - optimal mapping between ISA instructions and micro-instructions also varies over time
  - one-to-many mapping is not necessarily optimal
  - solution:
    - first brake down CISC instructions into micro-operations
    - than fuse micro-operations (from multiple CISC instructions) into micro-operations that fit pipeline better

Breaking the data flow limit (4)

- SIMD (single instruction multiple data)
  - observations
    - many (multimedia) applications operate on smaller data (8-bit, 16-bit)
    - executing them separately on 32-bit FUs is a waste of resources
  - original solution on RISC/VLIWs
    - adapt the FUs to do multiple narrow operations in parallel
    - add corresponding instructions to the ISA
    - add supporting instructions (packing, unpacking, shuffling, …)
    - are basically special-purpose instructions
    - some have very cheap implementation
      (How do you implement two 16-bit additions on 32-bit adders?)
  - was originally called subword parallelism

Breaking the data flow limit (5)

- SIMD cont'd
  - first solution on CISC (x86)
    - where in need of more registers anyway
    - but need backwards binary compatibility
    - add new MMX registers for MMX instructions only
    - so new registers besides new instructions
  - later extensions
    - even if we don't really need new registers
    - adding wider registers is a cheap way to optimize performance
      - complexity of most of pipeline stays the same
      - only part of the data path gets wider
    - SSEx (also for floating point, cryptography, ...)

Acknowledgement

- Thanks for (parts of) slides
  - Bjorn De Sutter
  - Lieven Eeckhout
  - Mikko H. Lipasti
  - James C. Hoe
  - John P. Shen