Processor pipelines revisited

From 5-stage to superscalar out-of-order processors

Introduction

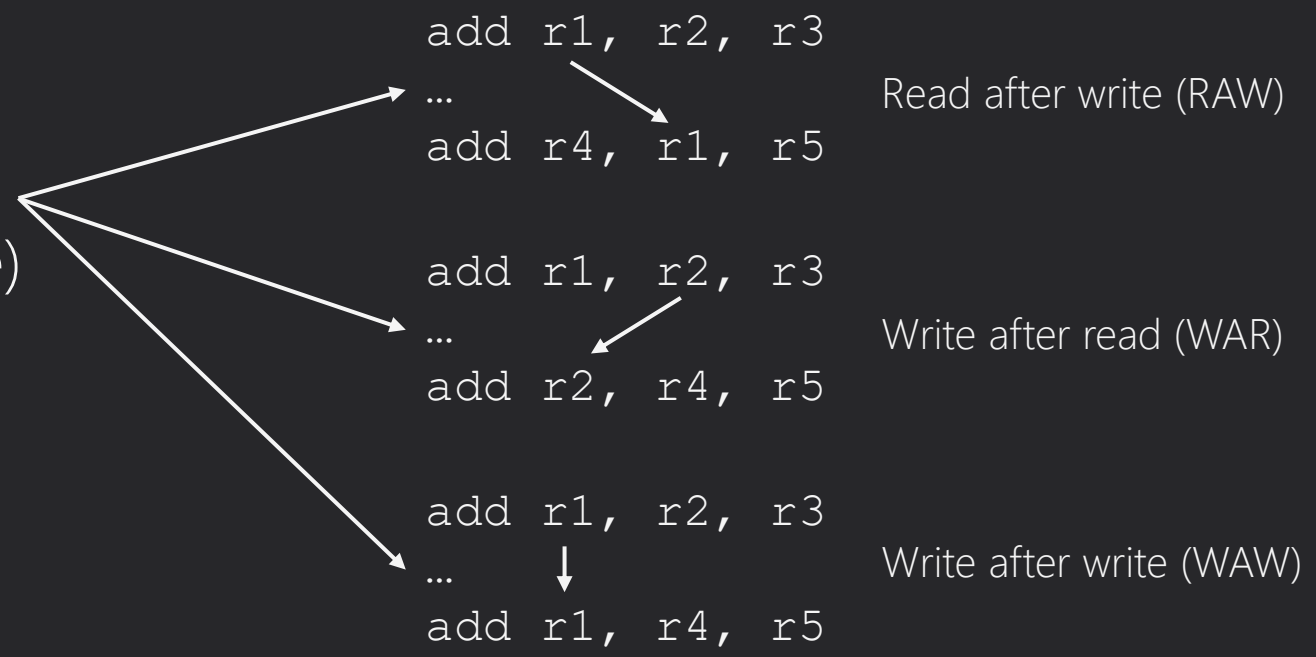
Scoreboards

Out-of-order execution

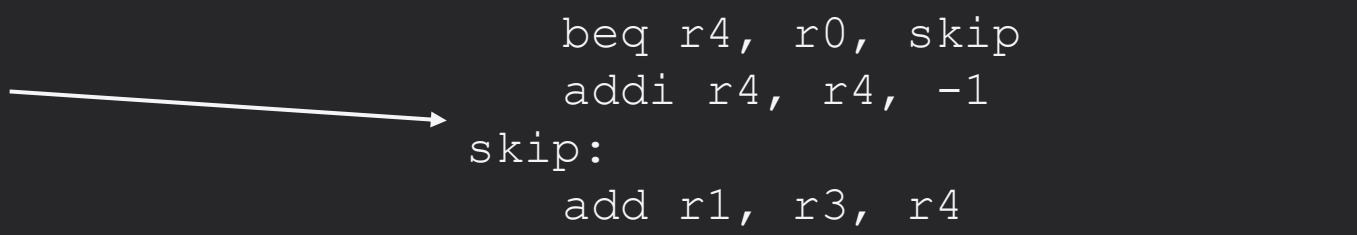
Conclusion

Dependencies

- Data
  - Two instructions use the same register (register data dependence)
  - Two instructions use the same memory location (memory data dependence)



- Control
  - One instruction decides whether another instruction will execute



Hazards

- When a dependence may impact correct execution
- Improper handling of hazards can corrupt data or control flow

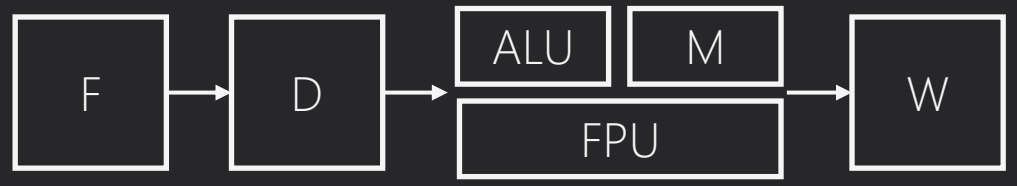
- Data
  - Solutions: stall, forwarding/bypassing
- Control
  - Solutions: stall, predict
- Structural
  - Solution: stall

The 5-stage pipeline

- Scalar pipeline: supports 1 instruction/stage
  - Ideal CPI = 1 (IPC = 1)

- Every instruction must go through every stage of the pipeline in-order
  - Some instructions only use a part of the pipeline
  - Stalls propagate backwards; trailing instructions in the pipeline must also stall

Pipeline diversification



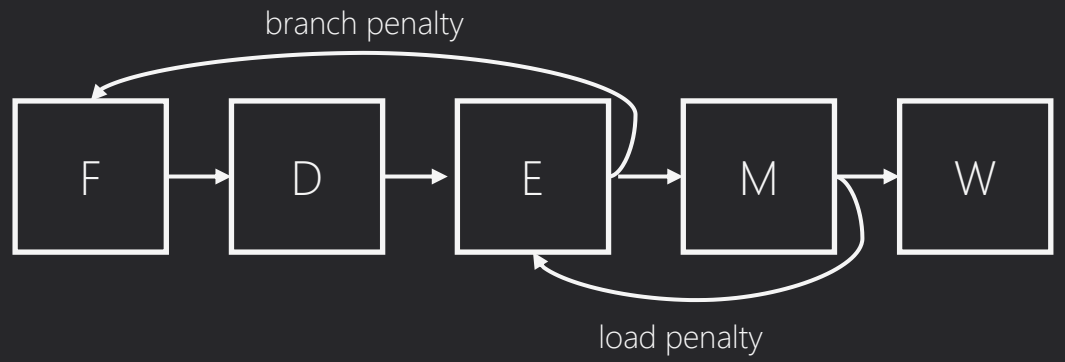
- Add diverse execution pipes (paths)
  - Different pipes for different instruction types
  - More efficient; each type of instruction has a specialized hardware

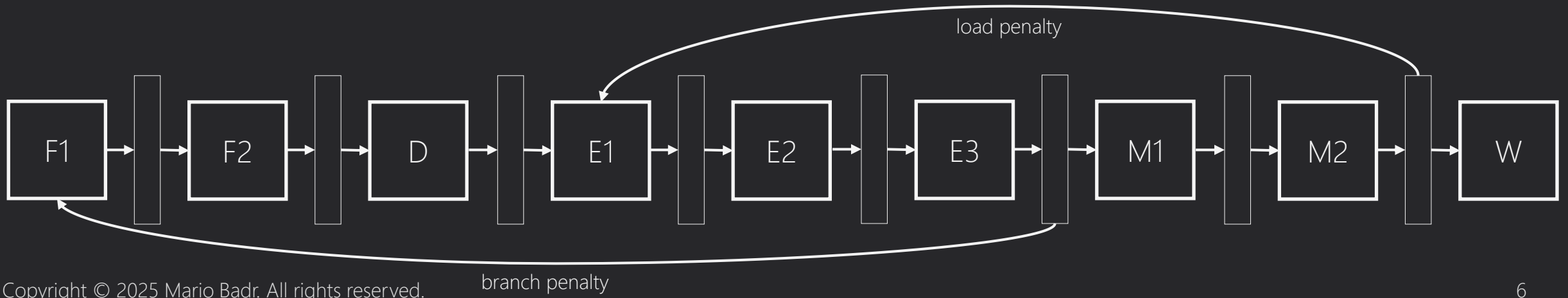
- But new problems are introduced…

	1	2	3	4	5	6	7	8	9	10
add.d f1, f2, f1	F	D	FP1	FP2	FP3	FP4	FP5	M	W	
add.d f4, f2, f3		F	D	FP1	FP2	FP3	FP4	FP5	M	W
l.d f10			F	D	E	M	W			
l.d f12				F	D	E	M	W		
l.d f14					F	D	E	M	W	

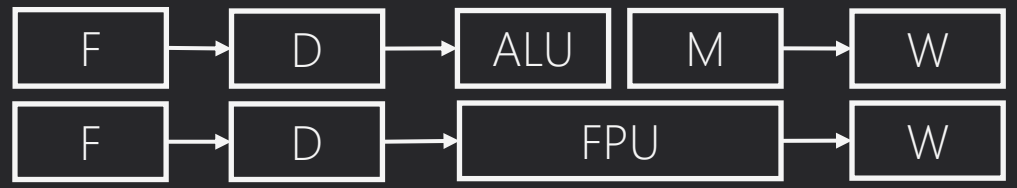
Deeper pipelines

- More stages reduce cycle time (+)
- More stages increase penalties due to pipeline hazards (-)





Wider pipelines



- Superscalar pipeline: work on up to m instructions per cycle
  - m = 2, 4, …
  - Typically, deeper and more diversified than a 5-stage pipeline

- Widening of the pipeline is expensive (and non-trivial)
  - Hardware cost (area, design complexity)
  - Instruction scheduling

The scoreboard

One way to schedule instructions

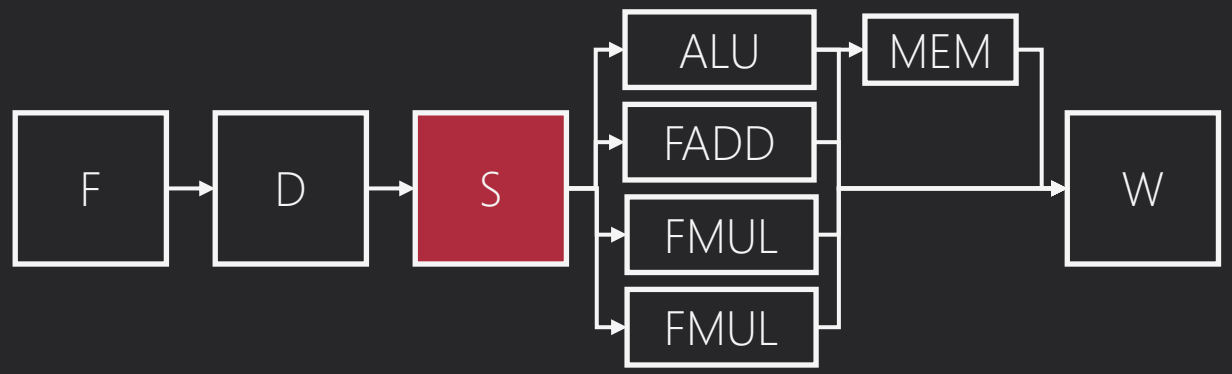
Instruction scheduling with a scoreboard

- A scoreboard is a *centralized* hardware scheduler
  - First seen in the CDC 6600 (1963)

- Instructions are sent from the front-end to the back-end when…
  - The functional unit is available
  - The inputs are available (RAW)
  - Writing to the output won't corrupt data (WAR, WAW)
  - Writing to the output won't conflict with another instruction (structural)

- The scheduler needs to keep track of scheduled instructions and functional units

An example pipeline



Functional Unit	Latency	Instructions
Int	1	e.g., add, ld
FADD	1	e.g., add.d
FMUL	3	e.g., mul.d
FMUL	3	e.g., mul.d

- The issue stage (S) schedules instructions to functional units
- Functional units read their operands on issue
- Instructions are issued in-order
  - Must stall on RAW, WAW
  - WAR is not possible (why?)

An example scoreboard

- Track each functional unit with 1 bit
  - 0: Functional unit is available
  - 1: Functional unit is busy
- Track whether a write is pending to each register
  - 0: No issued instruction is writing to this register
  - 1: A write is pending from an issued instruction
  - Updated in write back
- Assume no bypassing is available

Int	FADD	FMUL	FMUL
0	0	0	0

Register	Status
R0	
R1	
R2	
…	
R31	
F0	
F1	
…	

An example program

```
1. MUL.D F4, F0, F2
2. MUL.D F6, F4, F8
3. ADD.D F8, F2, F12
4. ADD.D F4, F14, F16
```

Int	FADD	FMUL	FMUL
0	0	0	0

Register	Status
…	
F0	0
F2	0
F4	0
F6	0
F8	0
F10	0
F12	0
F14	0
F16	0

Scoreboard example (cycle 1)

```
1. MUL.D F4, F0, F2    // FMUL (1)
2. MUL.D F6, F4, F8
3. ADD.D F8, F2, F12
4. ADD.D F4, F14, F16
```

Instruction 1 is issued to one of the available FMULs

- The status of the FMUL is updated to busy
- The FMUL doing the operation reads F0 and F2
- Register F4 is set to have a write pending

Int	FADD	FMUL	FMUL
0	0	1	0

Register	Status
…	
F0	0
F2	0
F4	1
F6	0
F8	0
F10	0
F12	0
F14	0
F16	0

Scoreboard example (cycle 2)

```
1. MUL.D F4, F0, F2    // FMUL (2)
2. MUL.D F6, F4, F8
3. ADD.D F8, F2, F12
4. ADD.D F4, F14, F16
```

Instruction 2 cannot be issued

- An FMUL is available (no structural)
- F6 (destination) does not have a write pending (no WAW)
- F4 (source operand 1) has a write pending (RAW!)
- F8 (source operand 2) does not have a write pending

Int	FADD	FMUL	FMUL
0	0	1	0

Register	Status
…	
F0	0
F2	0
F4	1
F6	0
F8	0
F10	0
F12	0
F14	0
F16	0

Scoreboard example (cycle 3)

```
1. MUL.D F4, F0, F2    // FMUL (3)
2. MUL.D F6, F4, F8
3. ADD.D F8, F2, F12
4. ADD.D F4, F14, F16
```

Waiting…

Int	FADD	FMUL	FMUL
0	0	1	0

Register	Status
…	
F0	0
F2	0
F4	1
F6	0
F8	0
F10	0
F12	0
F14	0
F16	0

Scoreboard example (cycle 4)

```
1. MUL.D F4, F0, F2    // Write back
2. MUL.D F6, F4, F8
3. ADD.D F8, F2, F12
4. ADD.D F4, F14, F16
```

Instruction 1 writes back result to F4

- The FMUL is no longer busy
- F4 no longer has a write pending

Instruction 2 still cannot issue (RAW)

Int	FADD	FMUL	FMUL
0	0	0	0

Register	Status
…	
F0	0
F2	0
F4	0
F6	0
F8	0
F10	0
F12	0
F14	0
F16	0

Scoreboard example (cycle 5)

```
1. MUL.D F4, F0, F2    // Done (@5)
2. MUL.D F6, F4, F8    // FMUL (1)
3. ADD.D F8, F2, F12
4. ADD.D F4, F14, F16
```

Int	FADD	FMUL	FMUL
0	0	1	0

Register	Status
…	
F0	0
F2	0
F4	0
F6	1
F8	0
F10	0
F12	0
F14	0
F16	0

Instruction 2 is issued to one of the available FMULs

- The status of the FMUL is updated to busy
- The FMUL doing the operation reads F4 and F8
- Register F6 is set to have a write pending

Scoreboard example (cycle 6)

```
1. MUL.D F4, F0, F2    // Done (@5)
2. MUL.D F6, F4, F8    // FMUL (2)
3. ADD.D F8, F2, F12   // FADD (1)
4. ADD.D F4, F14, F16
```

Instruction 3 is issued to FADD

- The status of the FADD is updated to busy
- The FADD reads F2 and F12
- Register F8 is set to have a write pending (No WAR)

Int	FADD	FMUL	FMUL
0	1	1	0

Register	Status
…	
F0	0
F2	0
F4	0
F6	1
F8	1
F10	0
F12	0
F14	0
F16	0

Scoreboard example (cycle 7)

```
1. MUL.D F4, F0, F2    // Done (@5)
2. MUL.D F6, F4, F8    // FMUL (3)
3. ADD.D F8, F2, F12   // Write back
4. ADD.D F4, F14, F16 // FADD (1)
```

Int	FADD	FMUL	FMUL
0	1	1	0

Register	Status
…	
F0	0
F2	0
F4	1
F6	1
F8	0
F10	0
F12	0
F14	0
F16	0

Instruction 3 writes back result to F8
- The FADD is no longer busy
- F8 no longer has a write pending

Instruction 4 is issued to FADD
- The status of the FADD is updated to busy
- The FADD reads F14 and F16
- Register F4 is set to have a write pending

(No WAR)

Scoreboard example (cycle 8)

```
1. MUL.D F4, F0, F2    // Done (@5)
2. MUL.D F6, F4, F8    // Write back
3. ADD.D F8, F2, F12   // Done (@8)
4. ADD.D F4, F14, F16 // Write back
```

Instruction 2 writes back result to F6
- The FMUL is no longer busy
- F6 no longer has a write pending

Instruction 4 cannot write back (structural hazard)
- The FADD is no longer busy
- We have assumed that write back can buffer the result until the register file's write port is available

Int	FADD	FMUL	FMUL
0	0	0	0

Register	Status
…	
F0	0
F2	0
F4	1
F6	0
F8	0
F10	0
F12	0
F14	0
F16	0

Scoreboard example (cycle 9)

```
1. MUL.D F4, F0, F2    // Done (@5)
2. MUL.D F6, F4, F8    // Done (@9)
3. ADD.D F8, F2, F12   // Done (@8)
4. ADD.D F4, F14, F16 // Write back
```

Instruction 4 writes back result to F4

- F4 no longer has a write pending

Int	FADD	FMUL	FMUL
0	0	0	0

Register	Status
…	
F0	0
F2	0
F4	0
F6	0
F8	0
F10	0
F12	0
F14	0
F16	0

Scoreboard summary

```
1.  MUL.D F4, F0, F2        // Done (@5)
2.  MUL.D F6, F4, F8        // Done (@9)
3.  ADD.D F8, F2, F12       // Done (@8)
4.  ADD.D F4, F14, F16      // Done (@10)
```

- Instructions can complete out-of-order (Instruction 3 before Instruction 2)
- RAW hazards still stall trailing instructions
  - e.g., instruction 3 did not have a RAW or WAW with Instructions 1 or 2, but had to wait
  - Limited by *in-order issue*

Int	FADD	FMUL	FMUL
0	0	0	0

Register	Status
…	
F0	0
F2	0
F4	0
F6	0
F8	0
F10	0
F12	0
F14	0
F16	0

Issuing out-of-order

The method

1. A buffer would hold all instructions waiting to issue
   - The instructions come from the decode stage
   - Decode ensures no WAW/WAR dependences
2. Instructions issue when there are no RAW hazards for the instruction
3. After write back, new instructions in the buffer may be schedulable

The implications

- Need to track more things in the scoreboard
  - i.e., now need to detect *WAR hazards*
- Performance improvement is still limited
  - … by control hazards (property of the software)
  - … by the number of registers (property of the ISA)

Register renaming

- In hardware

Register renaming eliminates WARs and WAWs

The idea

- Think of registers as *names* not physical locations
- Change a register's name to eliminate a WAR or WAW dependence
  - Leave RAW (*true dependencies*)
- We can have more physical locations than names

The implementation

- Need a data structure that maps a name to its (most recent) physical location
  - Write: allocate a name -> location mapping in the data structure
  - Read: use the name to look up the location
- Note: the data structure has a fixed size!
  - Need to de-allocate mappings to create room for new ones

An example of register renaming

```
1. div r1, r2, r3
2. add r4, r1, r5 # RAW (w/ 1)
3. add r5, r6, r7 # WAR (w/ 2)
4. add r1, r8, r9 # WAW (w/ 1)
```

After some register renaming,

```
1. div r32, r2, r3
2. add r33, r32, r5      # RAW (w/ 1)
3. add r34, r6, r7       # ~~WAR (w/ 2)~~
4. add r35, r8, r9       # ~~WAW (w/ 1)~~
```

- Logical registers
  - e.g., r0 to 31 (as defined by ISA)
- Physical registers
  - e.g., r0 to r63

- Logical registers r0 to r31 map to physical registers r0 to 31
- Leaves 32 other physical locations (r32 to r63)

The register renaming algorithm

```
1. div r1, r2, r3
2. add r4, r1, r5 # RAW (w/ 1)
3. add r5, r6, r7 # WAR (w/ 2)
4. add r1, r8, r9 # WAW (w/ 1)
```

After some register renaming,

```
1. div r32, r2, r3
2. add r33, r32, r5
3. add r34, r6, r7
4. add r35, r8, r9
```

	Register Mapping Table									*Free List*
	r1	*r2*	*r3*	*r4*	*r5*	*r6*	*r7*	*r8*	*r9*	
-	r1	r2	r3	r4	r5	r6	r7	r8	r9	r32 – r63
1	r32									r33 – r63
2				r33						r34 – r63
3					r34					r35-r63
4	r35									r36-r63

```
def renaming_stage(ri, rj, rk):
    # ri is dest, rj and rk are sources
    rb, rc = rename(rj), rename(rk)
    ra = freelist[first]

    rename(ri) = freelist[first]
    first = next(freelist)
    return (ra, rb, rc)
```

Storing results in program order

```
1. div r32, r2, r3        # r32 -> r1
2. add r33, r32, r5       # r33 -> r4
3. add r34, r6, r7        # r34 -> r5
4. add r35, r8, r9        # r35 -> r1
```

- What happens if instructions 3 and 4 compute their results before 1 and 2?
- Problem(s)
  - When can we write r34 to r5?
  - When can we write r35 to r1?
- Need a *reorder buffer*
  - Like a FIFO queue

Out-of-order scheduling

- With program-order writes

The re-order buffer

Extra steps in renaming algorithm:

```
ROB_tail = (0, None, ri, op)
ROB_tail = next(ROB_tail)
```

Instructions can *commit* only when they are at the head of the queue:

```
if (inst == ROB_head) and ROB_head.done):
        ri = value
        ROB_head = next(ROB_head)
```

Done?	*Value*	*Destination*	*Operation*

The re-order buffer over time (1)

Done?	*Value*	*Destination*	*Operation*	
0	None	r1	div	head
0	None	r4	add	
0	None	r5	add	
				tail

	Register Mapping Table									*Free List*
	r1	*r2*	*r3*	*r4*	*r5*	*r6*	*r7*	*r8*	*r9*	
-	r1	r2	r3	r4	r5	r6	r7	r8	r9	r32 – r63
1	r32									r33 – r63
2				r33						r34 – r63
3					r34					r35-r63

The first three instructions have been decoded and
1. their registers renamed
2. they are added to the ROB in program order

```
1. div r1, r2, r3
2. add r4, r1, r5
3. add r5, r6, r7
4. add r1, r8, r9
```

The re-order buffer over time (2)

Done?	*Value*	*Destination*	*Operation*	
0	None	r1	div	head
0	None	r4	add	
1	Something	r5	add	
1	Something	r1	add	
				tail

	Register Mapping Table									*Free List*
	r1	*r2*	*r3*	*r4*	*r5*	*r6*	*r7*	*r8*	*r9*	
-	r1	r2	r3	r4	r5	r6	r7	r8	r9	r32 – r63
1	r32									r33 – r63
2				r33						r34 – r63
3					r34					r35-r63
4	r35									

The fourth (and other) instruction was decoded and renamed earlier (not shown)

Instructions 3 and 4 have completed execution. But,
1. The results are in physical locations r34 and r35
2. They are not at the head of the ROB

```
1. div r1, r2, r3
2. add r4, r1, r5
3. add r5, r6, r7
4. add r1, r8, r9
```

The re-order buffer over time (3)

Done?	*Value*	*Destination*	*Operation*	
1	Something	r1	div	
0	None	r4	add	head
1	Something	r5	add	
1	Something	r1	add	
				tail

	Register Mapping Table									*Free List*
	r1	*r2*	*r3*	*r4*	*r5*	*r6*	*r7*	*r8*	*r9*	
-	r1	r2	r3	r4	r5	r6	r7	r8	r9	r32 – r63
1	r32									r33 – r63
2				r33						r34 – r63
3					r34					r35-r63
4	r35									

The first instruction completes.

1. The result from i1 (currently in r32) is written to r1

```
1. div r1, r2, r3
2. add r4, r1, r5
3. add r5, r6, r7
4. add r1, r8, r9
```

The re-order buffer over time (4)

Done?	*Value*	*Destination*	*Operation*	
1	Something	r1	div	
1	Something	r4	add	
1	Something	r5	add	head
1	Something	r1	add	
				tail

	Register Mapping Table									*Free List*
	r1	*r2*	*r3*	*r4*	*r5*	*r6*	*r7*	*r8*	*r9*	
-	r1	r2	r3	r4	r5	r6	r7	r8	r9	r32 – r63
1	r32									r33 – r63
2				r33						r34 – r63
3					r34					r35-r63
4	r35									

The second instruction completes.

Instructions 2, 3, and 4 are ready to commit.

Aside: In a superscalar pipeline, it would be ideal to be able to commit the same number of instructions as can be renamed in one cycle.

```
1. div r1, r2, r3
2. add r4, r1, r5
3. add r5, r6, r7
4. add r1, r8, r9
```

The reservation station

The problem(s)

- Where does an instruction wait?
- How does an instruction know it is ready to be executed?

The solution

- Wait in a reservation station (RS)
- A reservation contains things like,
  - The operation to be done
  - The values and names of source operands
  - The physical name of the result register
  - Where results should be stored in the ROB
- Scheduling is done based on a group of reservation stations
  - Called the *instruction window*
  - Need a scheduling policy (e.g., oldest instruction first)

An out-of-order pipeline

	Stage	Resources read	Resources written
Front-end	Fetch	PC Instruction cache	PC Instruction buffer
	Decode (rename)	Instruction buffer Register mapping table	Decode buffer Register mapping table
	Dispatch	Decode buffer Register mapping table Register file(s)	Reservation stations Re-order buffer
Back-end	Issue	Reservation stations	Functional units
	Execute	Functional units	Reservation stations Re-order buffer Physical register file
	Commit	Re-order buffer Physical register file	Re-order buffer Logical register file Register mapping table

The dispatch stage

High-level

- Attempts to assign instructions from the decode buffer to a reservation station
- If all reservation stations are being used, the front-end will stall
- Otherwise, fill the reservation station

Filling a reservation station

- Store the operation of the instruction
- If a source operand value is ready, then it is stored in the RS
  - Otherwise, the name is stored
- Store a pointer to the ROB entry

The issue stage

High-level

- Attempts to assign an instruction waiting in an RS to a functional unit
- An instruction waiting in RS is ready when:
  - The values for its source operands are available
  - i.e., we wait for RAW hazards

When (and how) are values communicated?

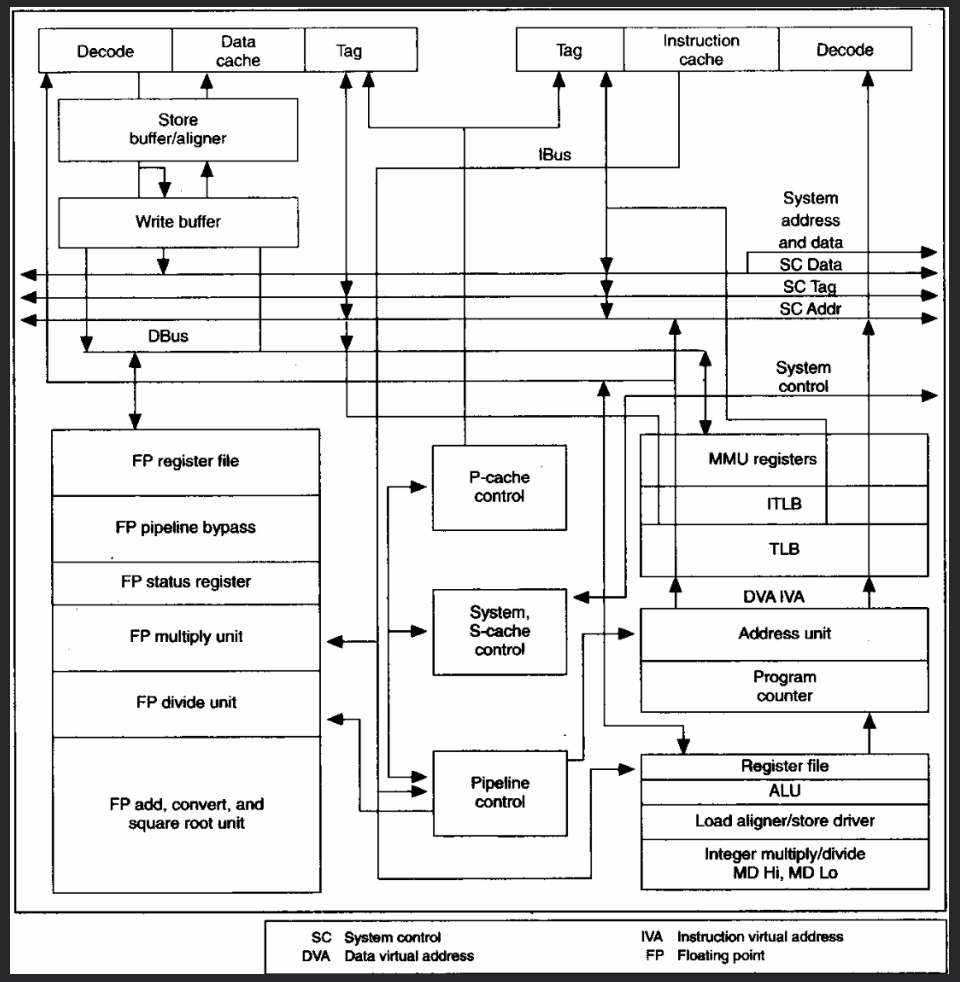
- When an operation completes, it broadcasts:
  - The name of the physical register
  - The value computed
- Broadcasts update:
  - Reservation stations that are waiting for the value
  - The physical register (e.g., in the ROB)

Conclusion

Connecting back to industry and academia

MIPS R4000

Mirapuri, Sunil, Michael Woodacre, and Nader Vasseghi. "The MIPS R4000 processor." IEEE Micro 12.2 (1992): 10-22.



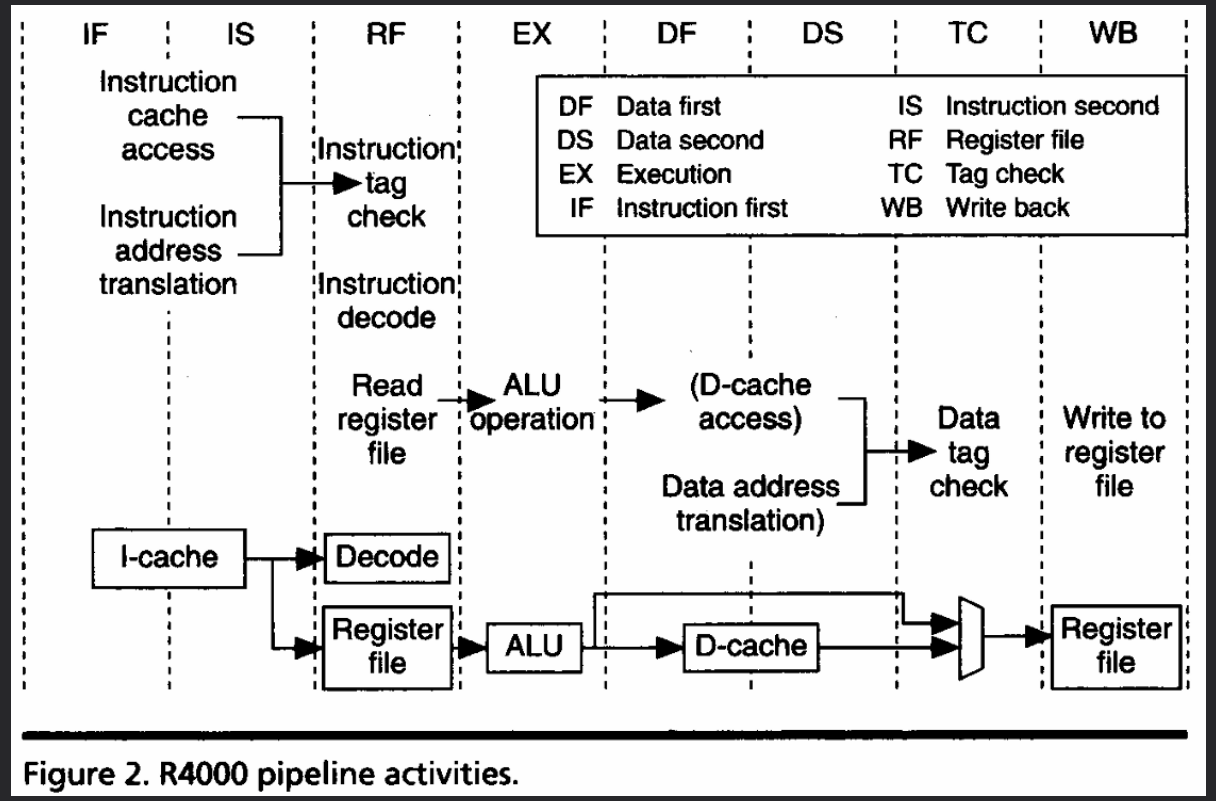
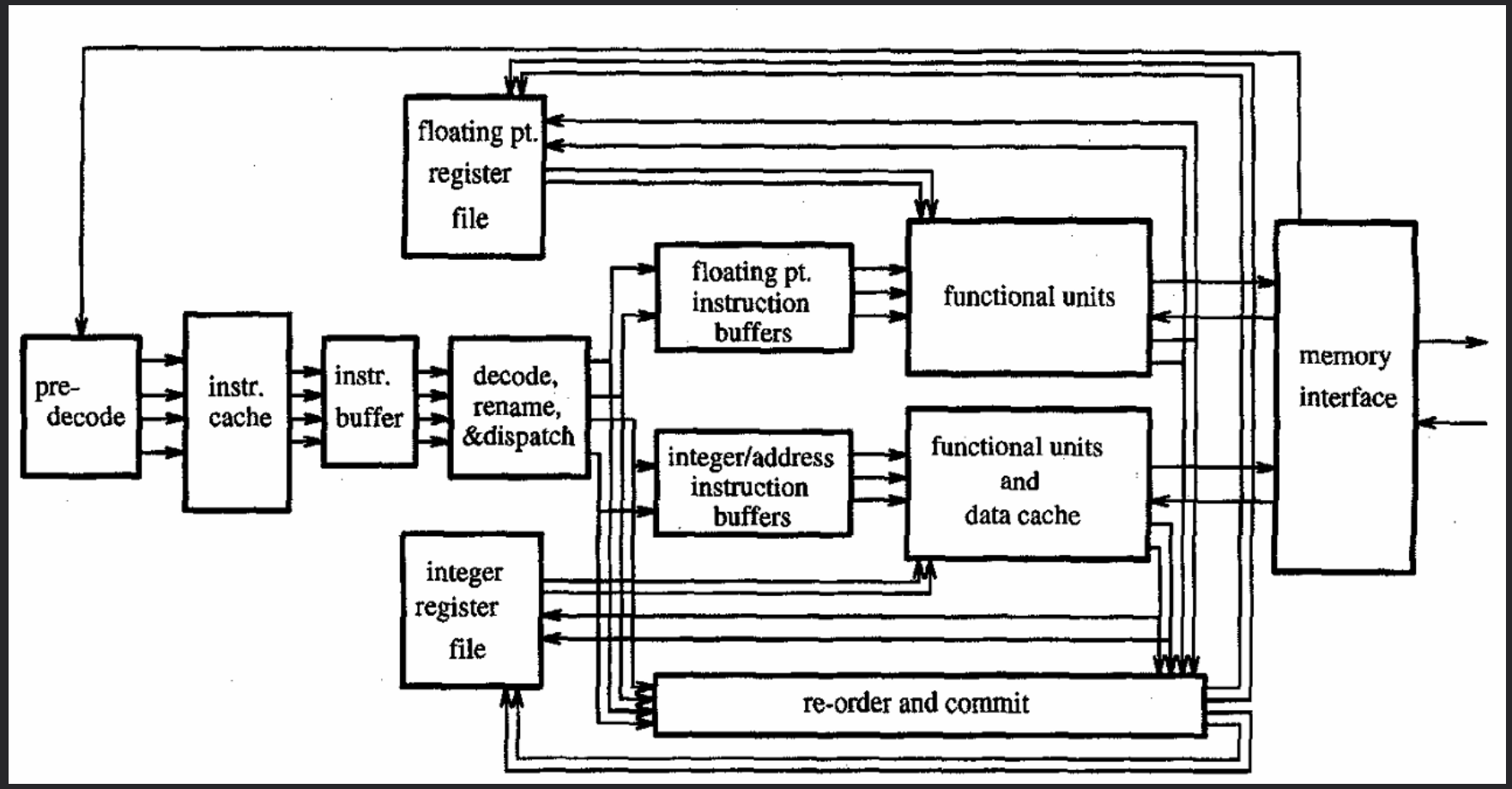


Figure 2. R4000 pipeline activities.

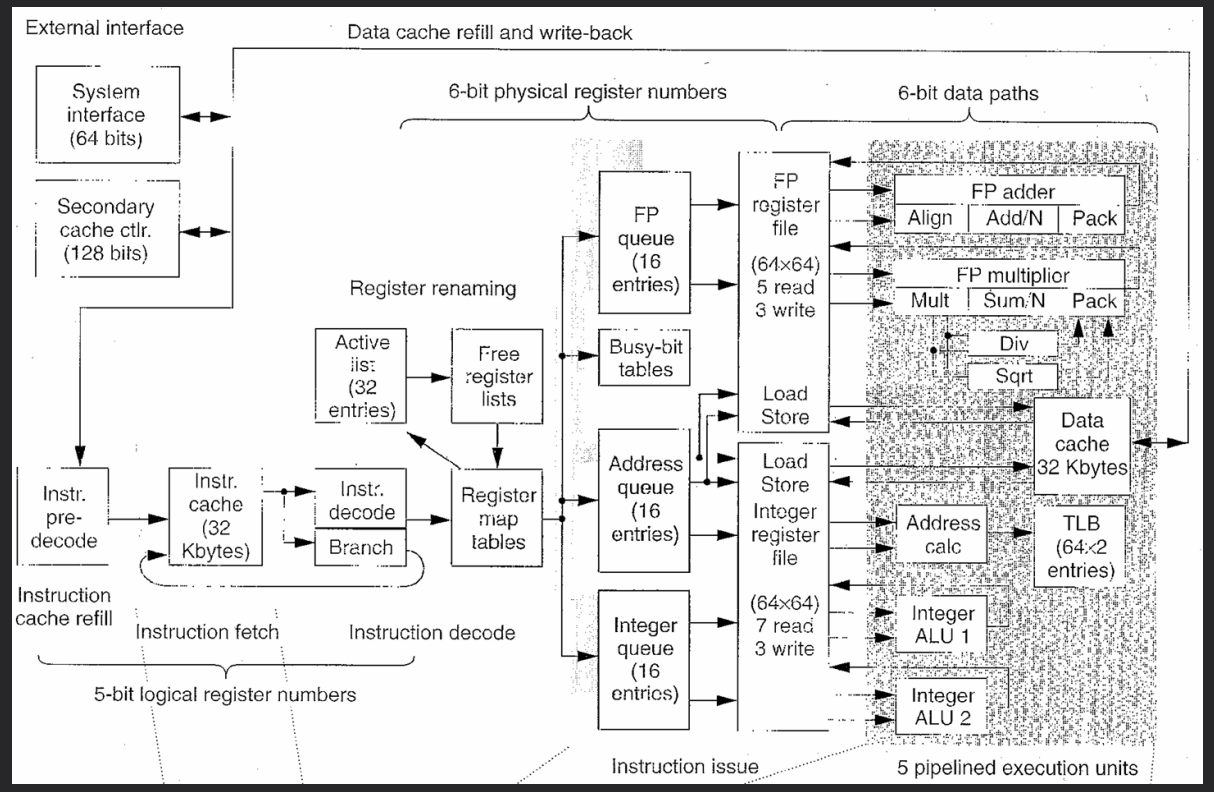
A “general” superscalar

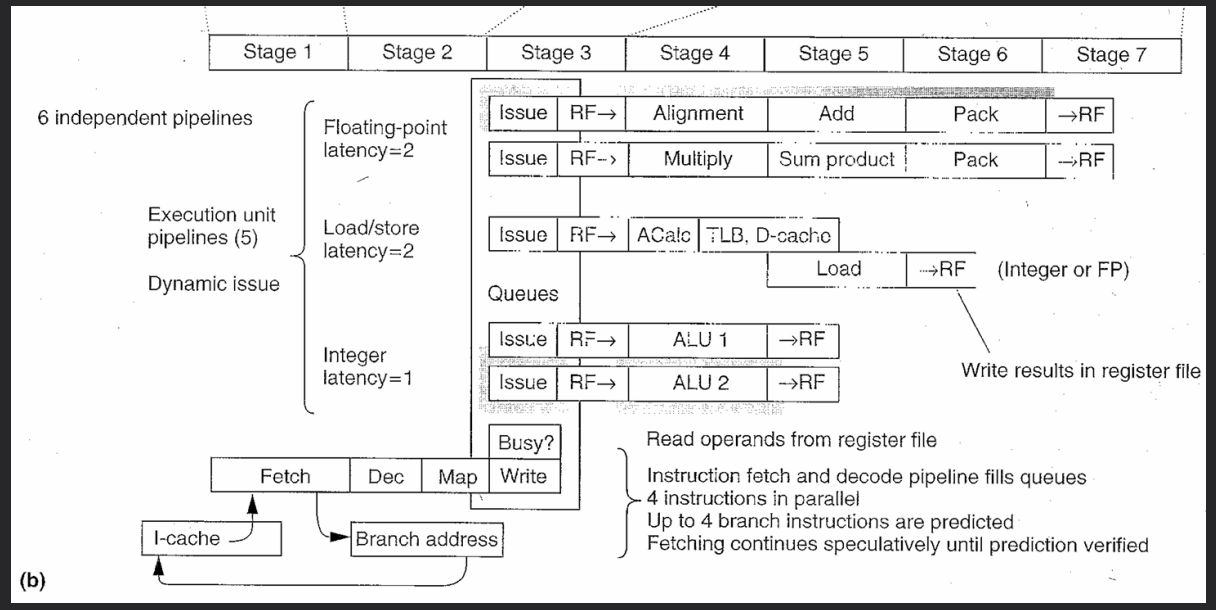
Smith, James E., and Gurindar S. Sohi. "The microarchitecture of superscalar processors." Proceedings of the IEEE 83.12 (1995): 1609-1624.



MIPS R10000

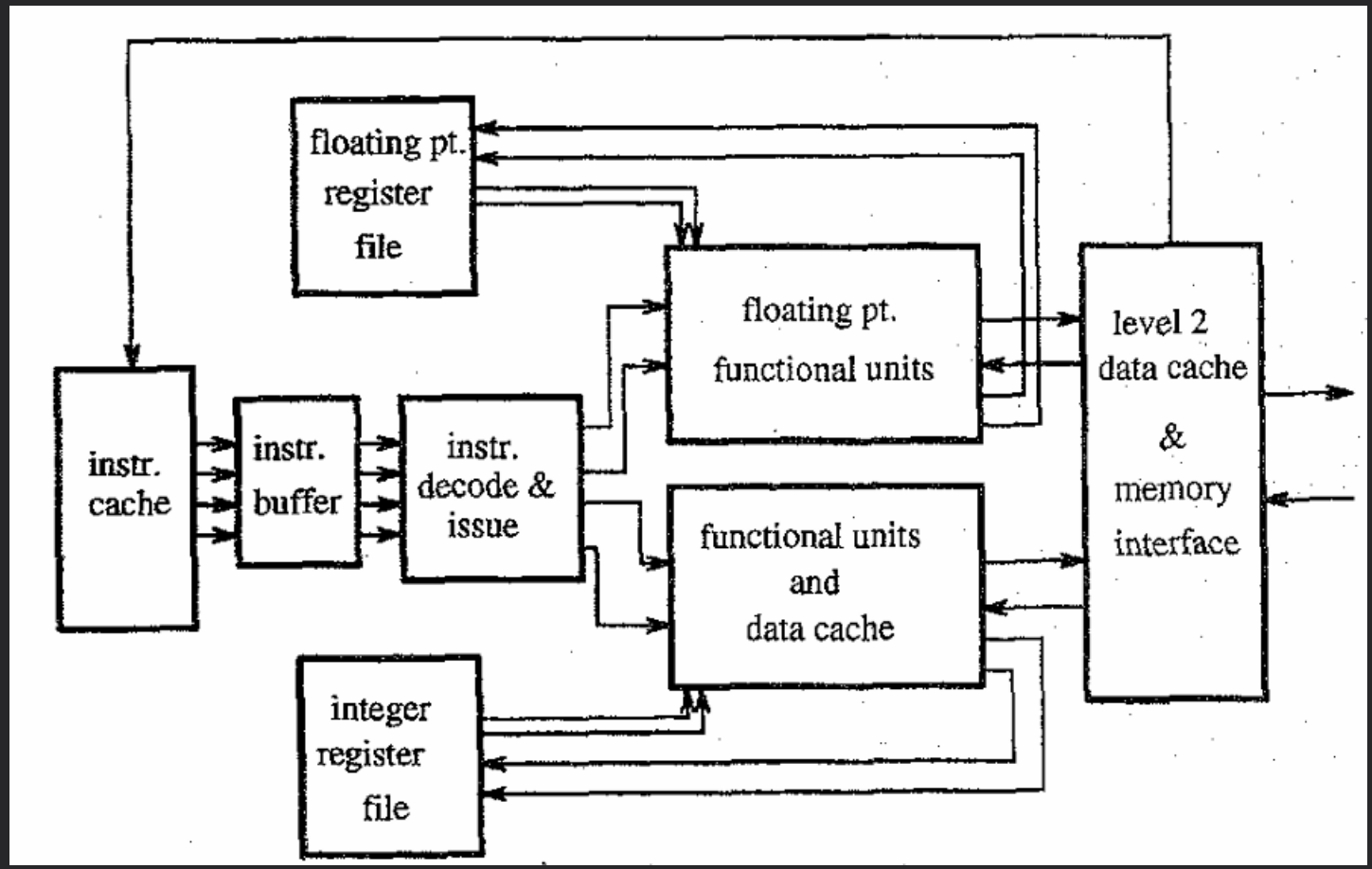
Yeager, Kenneth C. "The MIPS R10000 superscalar microprocessor." IEEE micro 16.2 (1996): 28-41.





DEC Alpha 21164

Smith, James E., and Gurindar S. Sohi. "The microarchitecture of superscalar processors." Proceedings of the IEEE 83.12 (1995): 1609-1624.



AMD K5

Smith, James E., and Gurindar S. Sohi. "The microarchitecture of superscalar processors." Proceedings of the IEEE 83.12 (1995): 1609-1624.