Computer Systems Design

Lesson 10

Data flow optimization

Alexander Antonov, Assoc. Prof., ITMO University

Hangzhou, 2025

ITMO

Outline the lesson

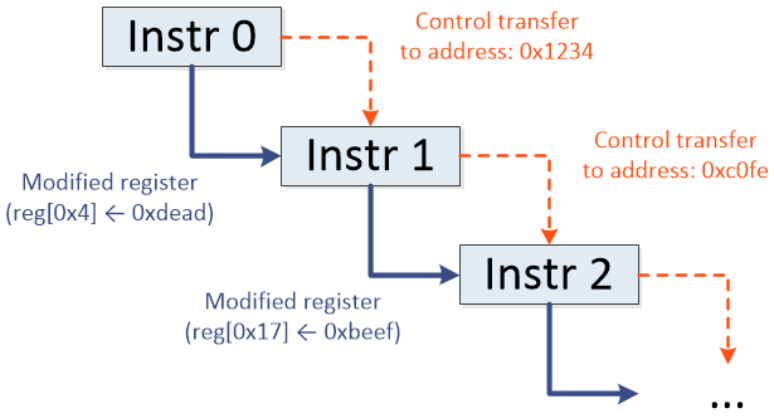
- Data flow optimization in classic RISC pipeline, pipeline hazards
- Data flow optimization in superscalar OoO microarchitectures
- HW and compiler optimizations
- Register renaming
- Value prediction

Instruction flow: control and data dependencies

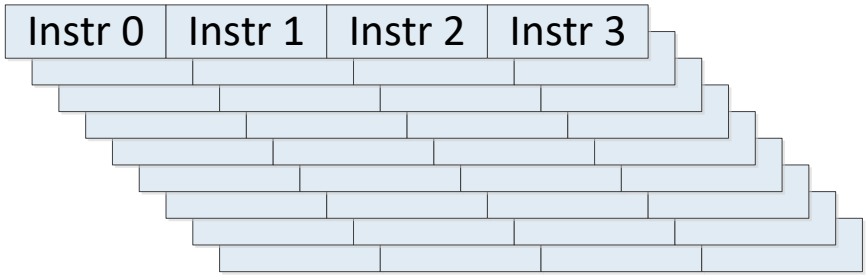
Each instruction (in instruction flow processors):

- modifies program state based on previously generated state ("data dependencies")
- passes control to some next instruction ("control dependencies")

Wanted: pipelined and parallel execution of instructions



Software-visible execution model



Actual (hardware) execution (pipelined and superscalar)

Pipeline hazards

Common types of hazards:

- ***Structural hazards***: access conflicts to the same hardware resources
  can be solved by hardware resources duplication
- ***Data hazards***: violation of program logic due to reordered read/write accesses to registers
- ***Control hazards***: violation of control flow due to overlapped instruction execution
  can be viewed as a form of data hazards applied to program counter

Pipeline hazards: concept

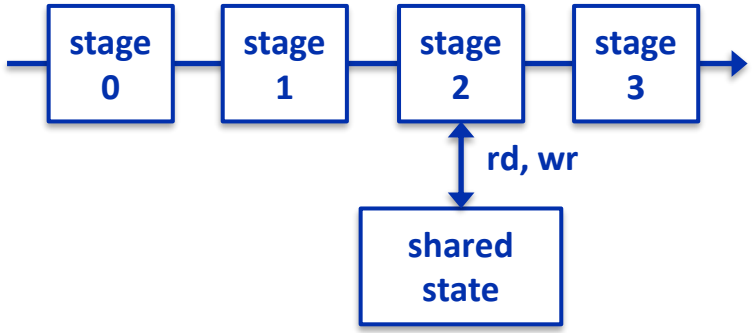
Problem of sequential → parallel instructions execution:

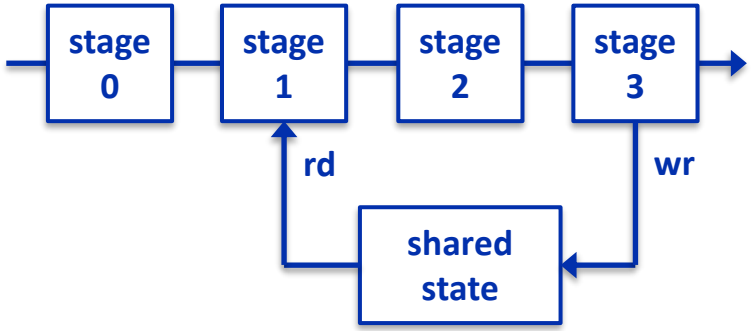
Changed ordering of working with shared states

Example: instruction flow

```
trx0: shared_state_v1 ← read (shared_state_v0) + 4;
trx1: shared_state_v2 ← read (shared_state_v1) + 8;
```

Expected result: `shared_state_v2 ← shared_state_v0 + 12;`





Operations sequence in hardware:

```
trx0: read (shared_state_v0);
trx0: shared_state_v1 ← read (shared_state_v0) + 4;
trx1: read (shared_state_v1);
trx1: shared_state_v2 ← read (shared_state_v1) + 8;
```

Result: shared_state_v0 + 12 (OK)

Operations sequence in hardware:

```
trx0: read (shared_state_v0);
trx1: read (shared_state_v0); // outdated, trx0 hasn't completed
trx0: shared_state_v1 ← read (shared_state_v0) + 4;
trx1: shared_state_v2 ← read (shared_state_v0) + 8;
```

Result: shared_state_v0 + 8 (not OK)

Data hazards classification: RAR, RAW

- **RAR** – read after read ☺

  «pseudo»-dependency

```
add r4, r6, r7
subi r8, r6, 0x100
```

Safe for reordering

- **RAW** – read after write

  «true» dependency

```
add r4, r6, r7
subi r8, r4, 0x100
```

Dangerous for reordering

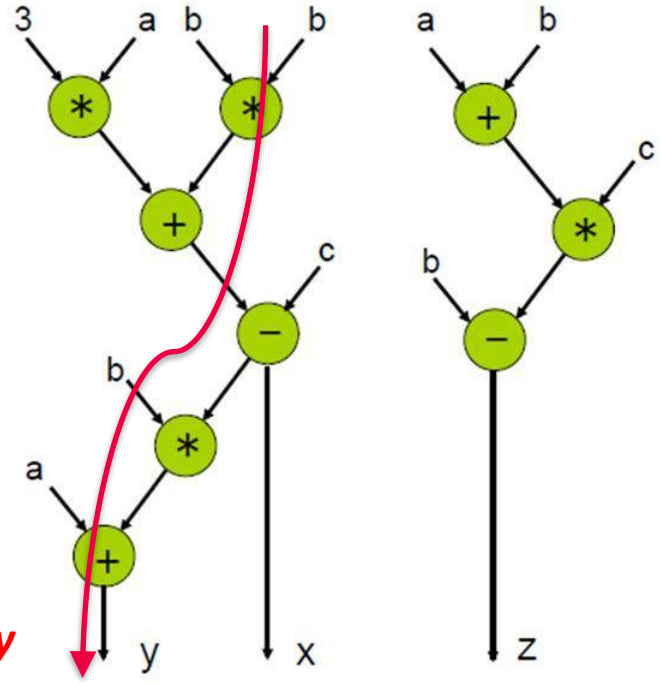
Data has to be computed before using

Visualization of RAW data dependencies: Data Flow Graph (DFG)

DFG (Data Flow Graph) – acyclic dependency graph between operations

Can be extracted from the ordered sequence of instructions

```
x = 3*a + b*b - c;
y = a + b*x;
z = b - c*(a + b);
```



Critical (longest) path: defines minimum latency ("dataflow limit")

Data hazards classification: WAR, WAW

- **WAR** – write after read

  «false» (anti-) dependency

```
add r4, r6, r7
subi r6, r8, 0x100
```

  Dangerous for reordering

  Can be solved is written data remapped to different register

- **WAW** – write after write

  «false» (output) dependency

```
add r8, r6, r7
subi r8, r4, 0x100
```

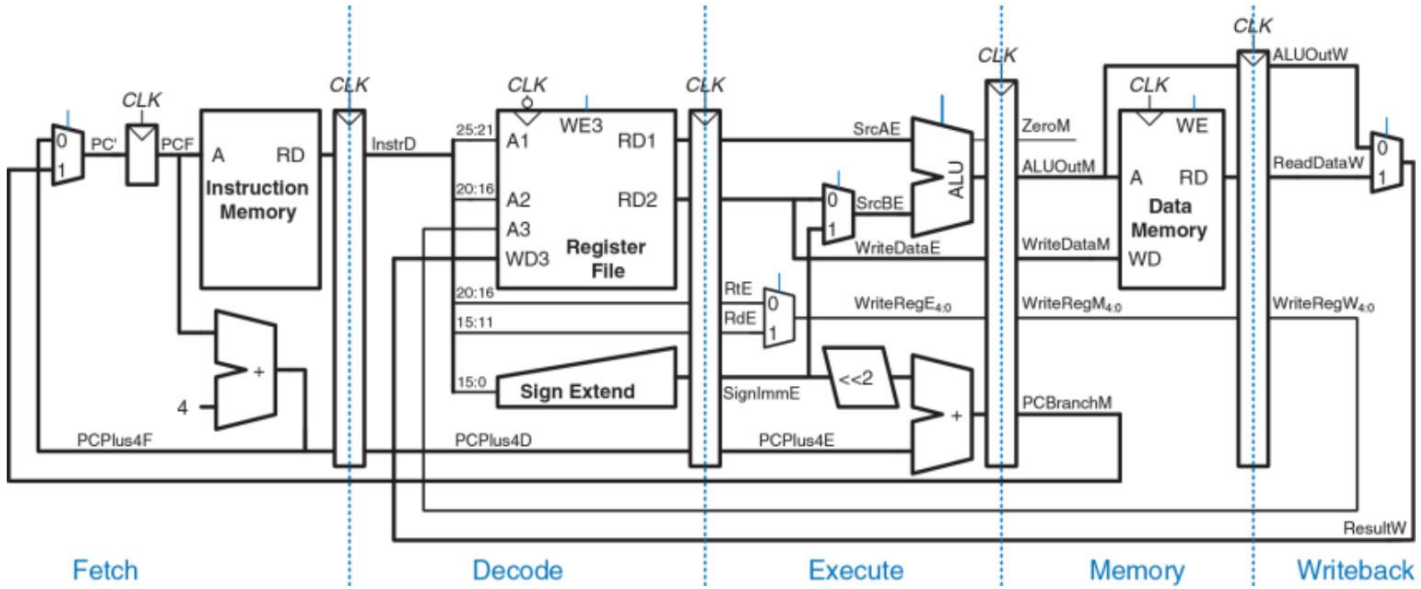
  Dangerous for reordering

  Can be solved is 2nd written data remapped to different register

Classic RISC pipeline

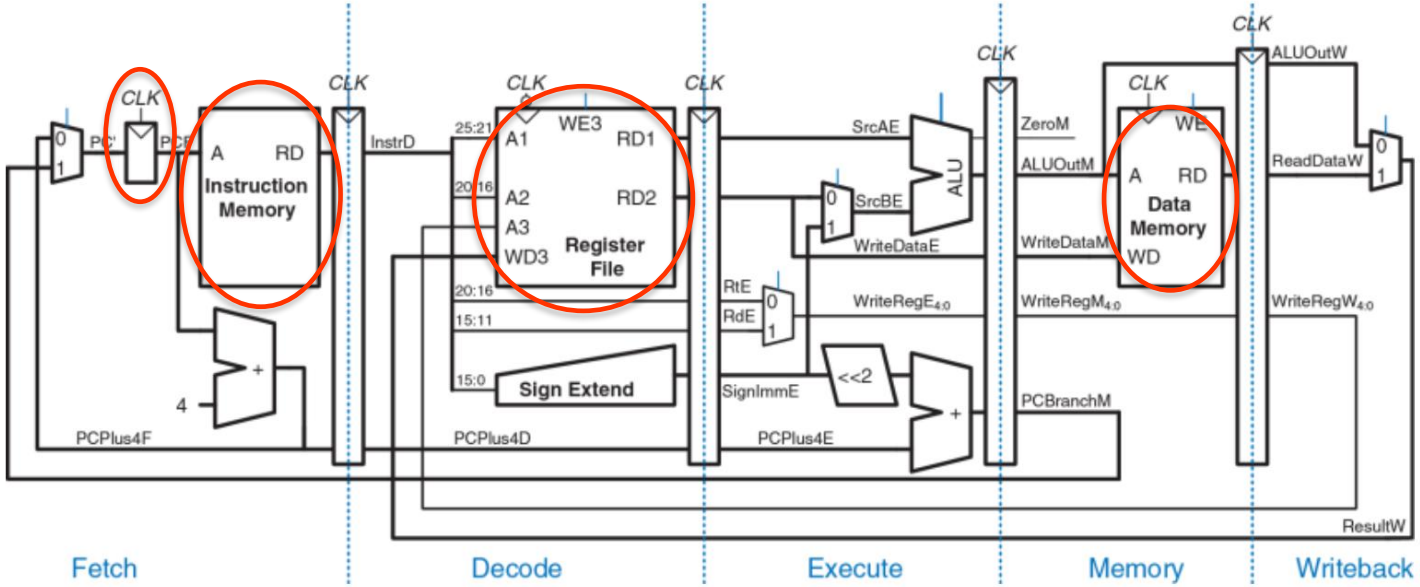
Vulnerability of arch states to hazards in classic RISC pipeline

Vulnerable architectural states?



Vulnerability of arch states to hazards in classic RISC pipeline

Vulnerable architectural states:



Vulnerability of arch states to hazards in classic RISC pipeline

Harvard – not dangerous:
- read at Fetch
- not written anywhere

Princeton (vN) – dangerous (**RAW**):
- read at Fetch
- written at Memory

Dangerous (**RAW**):
- read at Fetch
- written at Fetch and Execute

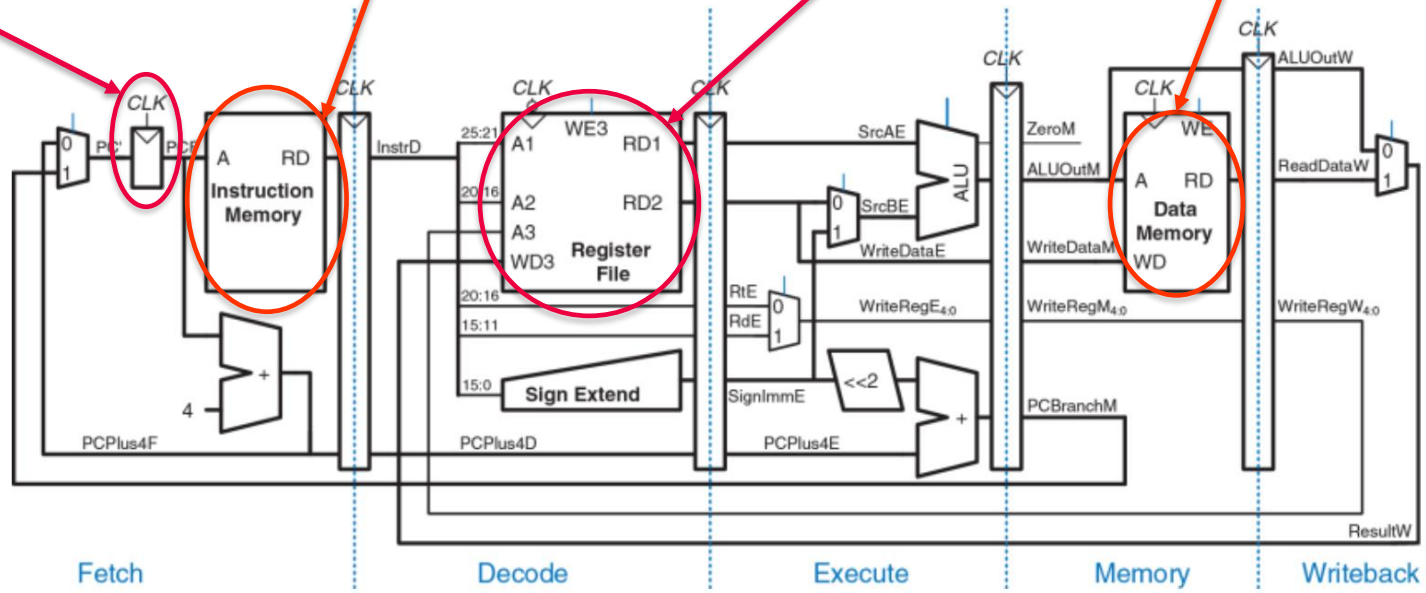
Referred to at “control hazards”

Dangerous (**RAW**):
- read at Decode
- written at Writeback

Referred to at “data hazards”

Harvard – not dangerous:
- read at Memory
- written at Memory

Princeton (vN) – dangerous (**RAW**):
- read at Fetch
- written at Memory



ITMO

Hazards: registers vs. PC

Hazard aspect	Registers	PC
Occurrence		

ITMO

Hazards: registers vs. PC

Hazard aspect	Registers	PC
Occurrence	Sometimes (if reading register that will be written by in-flight instruction)	Always (instruction always passes control to some next instruction)
Timing of reading		

ITMO

Hazards: registers vs. PC

Hazard aspect	Registers	PC
Occurrence	Sometimes (if reading register that will be written by in-flight instruction)	Always (instruction always passes control to some next instruction)
Timing of reading	In the middle of the pipeline: after instruction is fetched and decoded but before execution	Early (for fetching instruction code). Defines all further processing of instruction.
Timing of generation		

ITMO

Hazards: registers vs. PC

Hazard aspect	Registers	PC
Occurrence	Sometimes (if reading register that will be written by in-flight instruction)	Always (instruction always passes control to some next instruction)
Timing of reading	In the middle of the pipeline: after instruction is fetched and decoded but before execution	Early (for fetching instruction code). Defines all further processing of instruction.
Timing of generation	Late (on writeback stage, after execution on corresponding FU)	Variable (instruction can be non-branch, unconditional branch, worst case: conditional computed)
Value predictability		

ITMO

Hazards: registers vs. PC

Hazard aspect	Registers	PC
Occurrence	Sometimes (if reading register that will be written by in-flight instruction)	Always (instruction always passes control to some next instruction)
Timing of reading	In the middle of the pipeline: after instruction is fetched and decoded but before execution	Early (for fetching instruction code). Defines all further processing of instruction.
Timing of generation	Late (on writeback stage, after execution on corresponding FU)	Variable (instruction can be non-branch, unconditional branch, worst case: conditional computed)
Value predictability	Bad. Continuously recomputed in most cases	Good (>99%), based on previous outcomes, branch targets, …

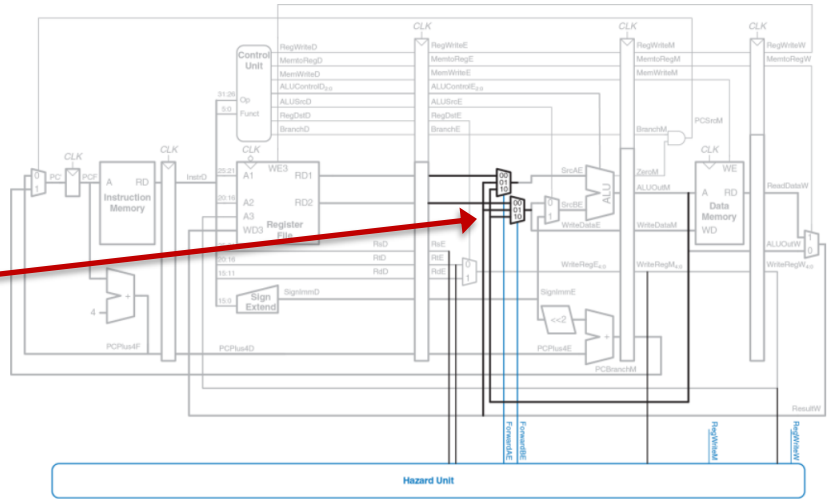
RAW register data hazard resolution in classic RISC pipeline

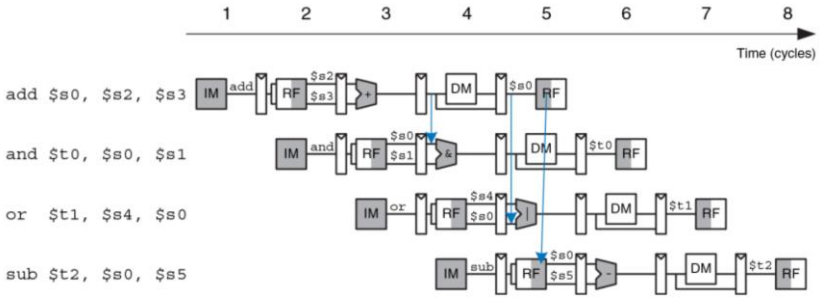
- Solution 1: ***identify*** RAW hazard (by register address) and ***stall*** until data is written
- Solution 2: ***identify*** RAW hazard (by register address) and ***forward*** data (if ready) from producer directly to consumer

Stalls can happen anyway if instruction requires data from earlier "slow" operation

e.g. memory, mul/div, FPU

- Solution 3: ***predict*** value, verify once available, discard is mispredicted





Bad: stalled instructions block the entire instruction flow

Bad: external fragmentation, lim (IPC) = 1

Static (compiler) RAW dependencies relaxation

Reorders code to make ***dependent instructions far from each other***

Obligatory if HW doesn't do data hazard resolution to ***ensure correctness*** (obsolete)

Optional (but useful) if HW does data hazard resolution to ***reduce stalls and improve performance***

```
lw r2, r1, 0x100
add r4, r2, r0
lw r5, r1, 0x104
subi r6, r5, 0x20
```

→

```
lw r2, r1, 0x100
lw r5, r1, 0x104
add r4, r2, r0
subi r6, r5, 0x20
```

GCC:

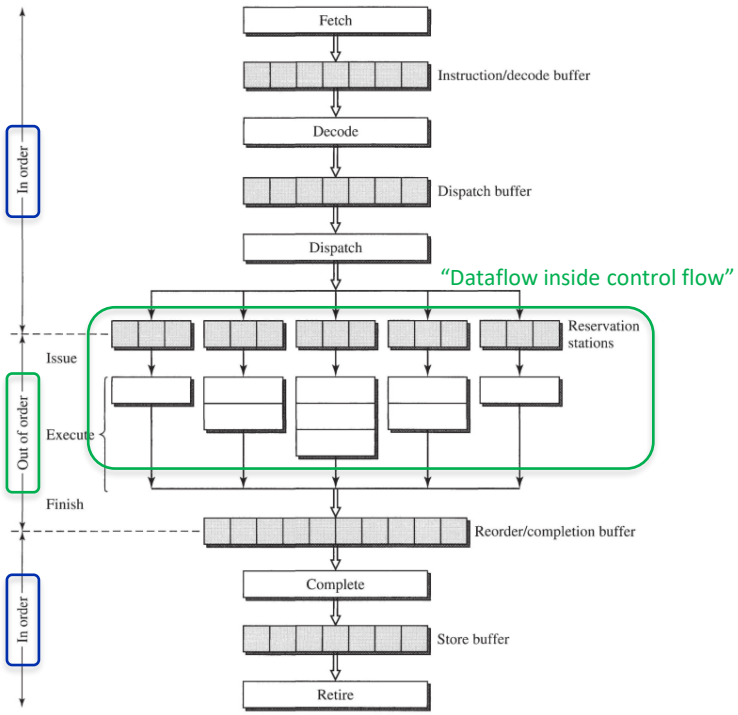
`-fschedule-insns`

If supported for the target machine, attempt to reorder instructions to eliminate execution stalls due to required data being unavailable. This helps machines that have slow floating point or memory load instructions by allowing other instructions to be issued until the result of the load or floating-point instruction is required.

Enabled at levels `-O2`, `-O3`.

Superscalar OoO microarchitectures

Out-of-order execution in superscalar CPUs: scoreboarding (CDC 6600, 1965 …)



Instruction status				
Instruction	Waiting for issue	Waiting for data	Executing	Writing result
`add dst, sr0, src1`	N	N	N	Y
`mul dst, sr0, src1`	N	N	Y	N
…	…	…	…	…
`jmp condition label`	Y	N	N	N

Register status	
Register number	FU number
Reg 0	FU 0
Reg 1	FU 5
…	…
Reg N	FU 3

FU status									
FU number	Busy flag	Opcode	Src reg 0	Src reg 1	Dst reg	FU src 0	FU src 1	Src 0 ready	Src 1 ready
0	Y	add	0xa	0xb	0x4	FU 2	FU 3	Y	N
1	N	sub	0x3	0x1	0x8	FU 6	FU 4	N	Y
…	…	…	…	…	…	…	…	…	…
2	Y	mul	0x10	0x7	0x2	FU 7	FU 1	Y	Y

Good: stalled instructions <u>don't block instruction flow</u>

Good: <u>less external fragmentation (diverse pipelines)</u>, lim (IPC) ≠ 1

Bad: <u>WAR and WAW hazards possible</u>, registers should be marked as busy on writing instruction until write is done

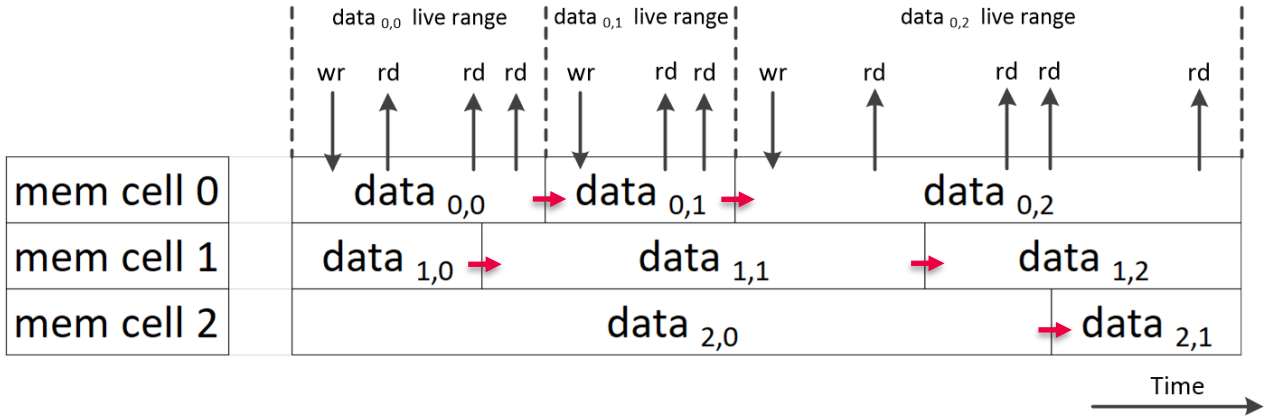
Picture: Shen, J. P., & Lipasti, M. H. Modern processor design: fundamentals of superscalar processors. Waveland Press, 2013

Memory cell vs. immutable data instance

Both: identifiable (addressable) states

Difference:

- **Memory cell life cycle (mutable):** repeating reads and writes
- **Data instance life cycle (immutable):** single initialization (on write) and repeating usage (reads) until rewrite

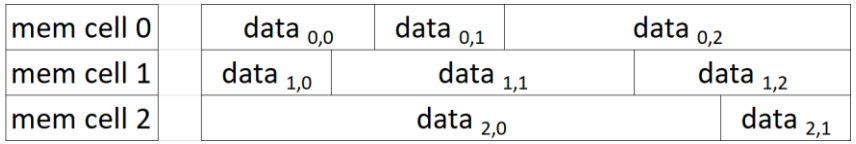


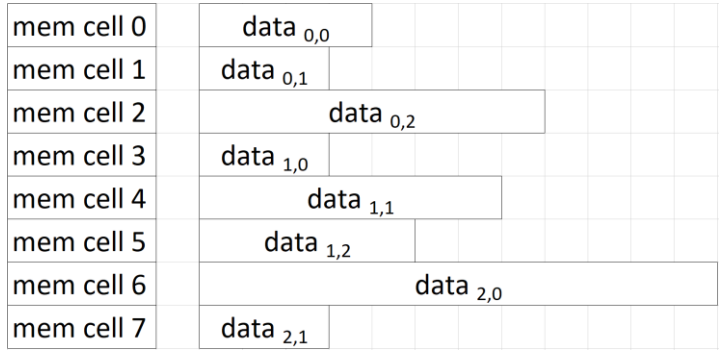
Memory cells are reused for various data because of finite number of cells

Restricts parallelism because of false dependencies (WAR, WAW)

Parallelism can be restored if data is remapped from single cell to multiple cells

Memory footprint vs. parallelism: trade-off





- Smaller memory footprint
- More false dependencies
- Less explicit parallelism

SSA
Register renaming

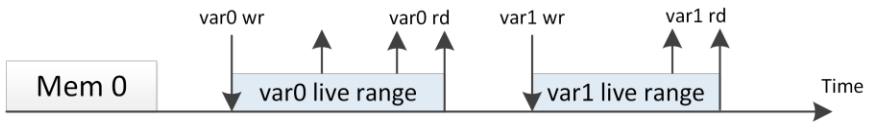
Register recycling (reuse)
Memory overlays

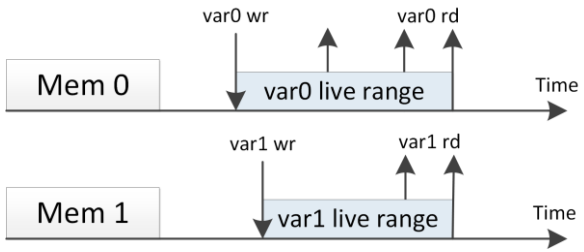
- Larger memory footprint
- Less false dependencies
- More explicit parallelism

Compilers: static false dependencies resolution

Various data instances can be ***remapped on different memory cells***

Potential parallelism increases (***false data dependencies removed***)





Compiler IRs can ***abstract finite memory*** (offer ***infinite number of registers***)

"virtual registers", "pseudo-registers" (GCC, LLVM)

False data dependencies (mostly) eliminated in ***Single Static Assignment*** (***SSA***) form

```
y := arg0 + 5
x := y % 4
y := arg2 * 3
x := y / 7
```

```
y1 := arg0 + 5
x1 := y1 % 4
y2 := arg2 * 3
x2 := y2 / 7
```

Dynamic false dependencies resolution: register renaming

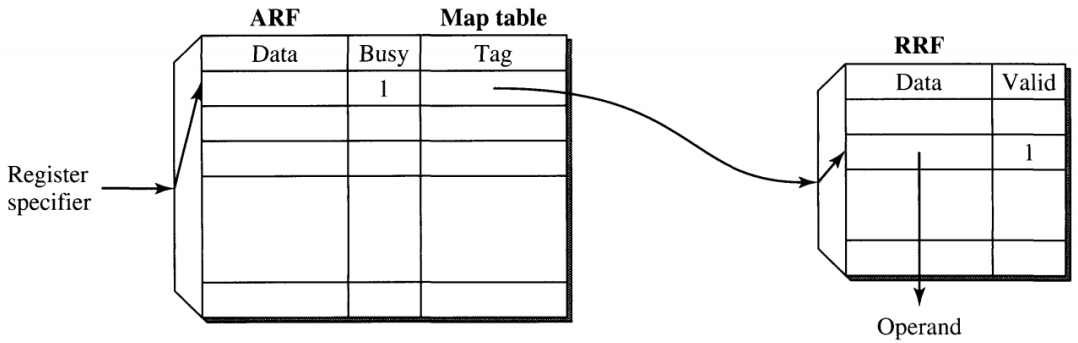
Remapping capabilities may **vary** depending on hardware resources

SSA **cannot handle dynamic instruction flow**:

- *not-unrolled loops*
- *loops with dynamic iterations control*
- *non-inlined function calls*

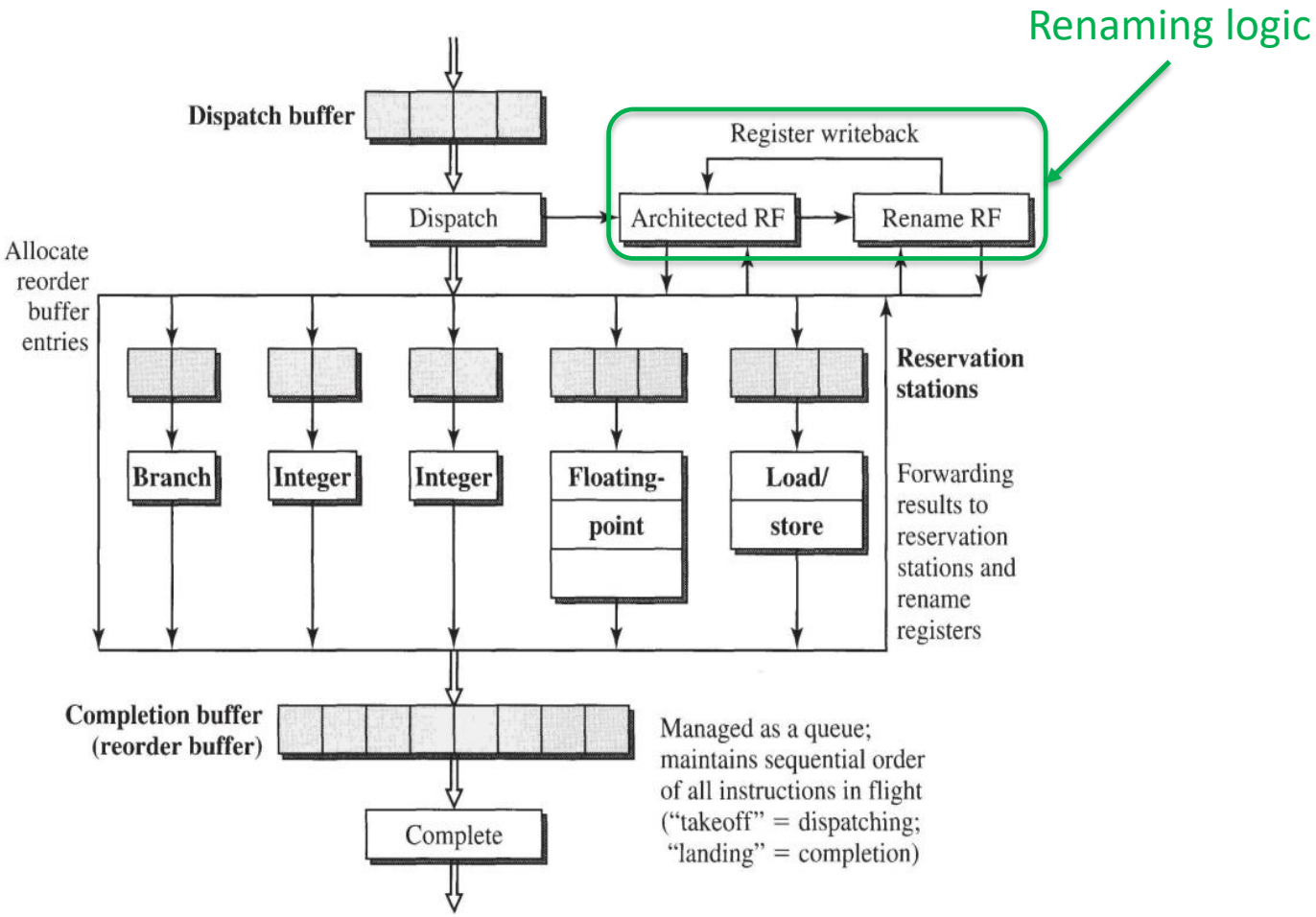
Dynamic false dependencies resolution required

Register file can be split in «architectural» (visible to software) and «physical» (with register addresses renamed to **tags** according to hardware capabilities)

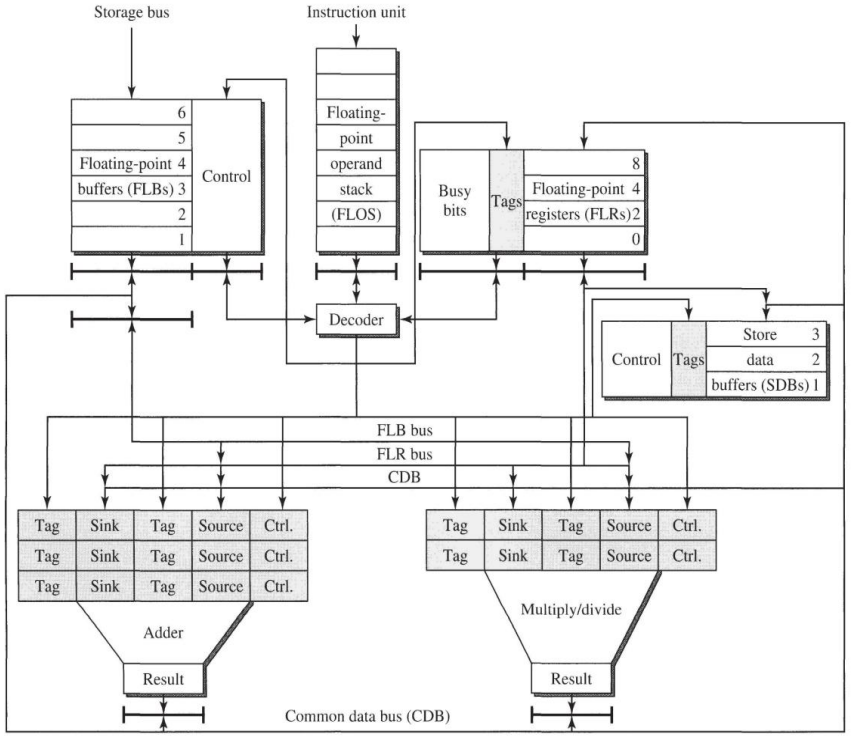


If enough physical memory cells – allows to reach dataflow limit (latency = DFG critical path)

CPU pipeline with register renaming



Classic renaming example: Tomasulo algorithm (IBM System/360 Model 91 FPU, 1967)



R.M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," IBM J. Res. Dev., no. January, pp. 25–33, 1967.

ITMO

Summary: typical register requests handling in various CPU microarchitectures

CPU uarch	Register read	Register write
Classic in-order RISC pipeline	Check for register readiness (**RAW hazards**). Stall, bypass, or predict value in case data is outdated	Write register data on WriteBack stage
Out-of-order (without renaming)	Check for register readiness (**RAW hazards**). Stall, bypass, or predict value in case data is outdated	Check for register readiness (**WAR, WAW hazards**), stall in case it's not ready. Data should be written to arch state only after instruction ordering restored and instruction completes successfully
Out-of-order (with renaming)	Check for physical register readiness (**RAW hazards**). Stall, bypass, or predict value in case register is not ready	Reserve new physical register from free register pool. Stall in case physical register pool is full. Data should be written to arch state only after instruction ordering restored and instruction completes successfully

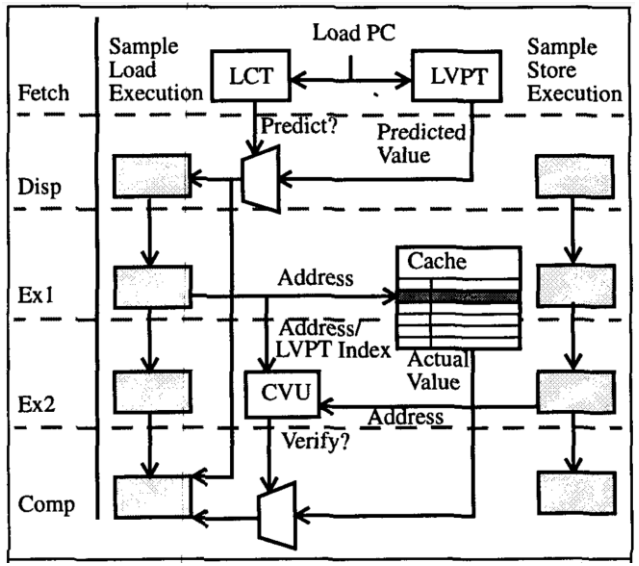
Data (value) prediction (1996)

Techniques attempting to ***break true (RAW) data dependencies***

Predict values based on previous ones ***prior to actual fetch/computation***

some re-computed/re-fetched data in registers actually remains constant or changes infrequently

- LCT (Load Classification Table)
  saturating counter classifying predictability of load
- LVPT (Load Value Prediction Table)
  contains predicted values for load instructions
- CVU (Constant Verification Unit)
  checker if value is actually constant (avoiding memory access)



Unknown to be implemented in commercial designs

Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. 1996. Value locality and load value prediction. SIGPLAN Not. 31, 9 (Sept. 1996), 138–147.

Thank you for the lesson!

Alexander Antonov, Assoc. Prof., antonov@itmo.ru

Hangzhou, 2025