CSC 411

Computer Organization (Fall 2024)
Lecture 23: Hazards and branch prediction

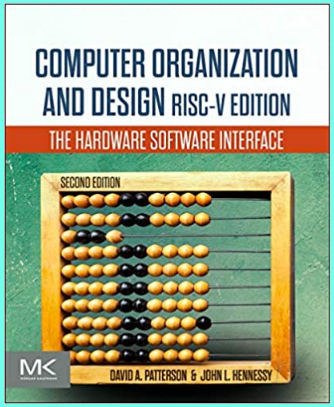
Prof. Marco Alvarez, University of Rhode Island

Disclaimer

Some figures and slides are adapted from:

Computer Organization and Design (Patterson and Hennessy)

The Hardware/Software Interface


COMPUTER ORGANIZATION AND DESIGN RISC-V EDITION
THE HARDWARE SOFTWARE INTERFACE
SECOND EDITION
MK
DAVID A. PATTERSON & JOHN L. HENNESSY

Hazards

Hazards

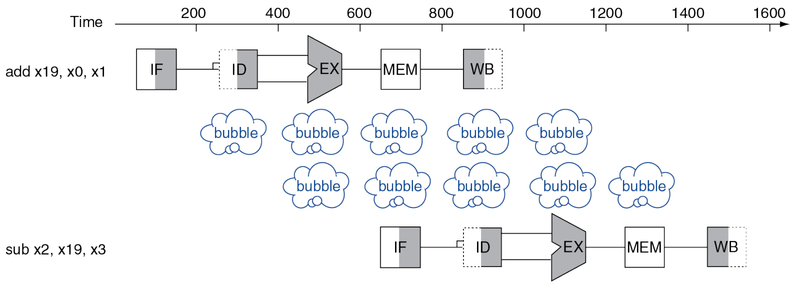
- Pipeline hazards that prevent sequential instruction execution
  - **Structural hazards**
    - resource conflicts when multiple instructions require the same hardware component simultaneously, examples include memory ports or execution units
  - **Data hazard**
    - dependencies where an instruction requires data produced by a previous instruction that has not yet completed
  - **Control hazard**
    - occurs when the instruction flow depends on a branch decision not yet resolved, pipeline must wait for branch condition evaluation before determining next instruction

Structural hazards

- Conflict for using the same resource
- e.g., RISC-V pipeline with a single memory
    - load/store requires data access
    - instruction fetch would have to **stall** for that cycle, causing a pipeline "*bubble*"
    - pipelined datapaths require separate instruction/data memories (separate instruction/data caches)

Data hazards

- An instruction depends on completion of data produced by a previous instruction

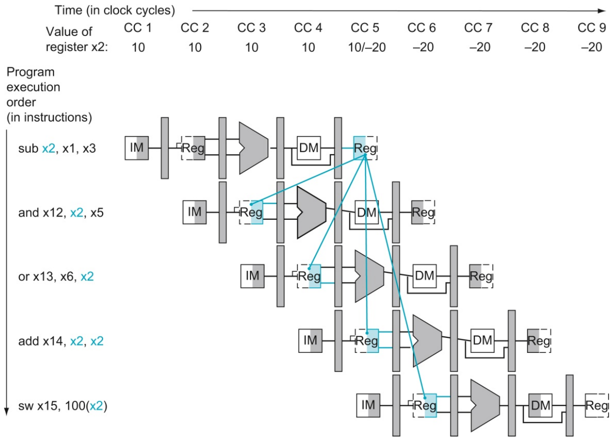


x19 value is produced in the WB stage, and the sub instruction requires the value in the ID stage. Two stall cycles are introduced to ensure correct execution.

Stalls and flushes

- Pipeline stall (also called *bubble*)
    - 1-cycle delay in pipeline execution where instructions are paused
    - implementation:
        - **holding current instruction values** in pipeline registers
        - inserting "**no operation**" (nop) into the pipeline
- Pipeline flush
    - discarding instructions from pipeline stages
        - clears all instructions, resets all control signals, and restarts instruction fetch

Pipeline dependencies

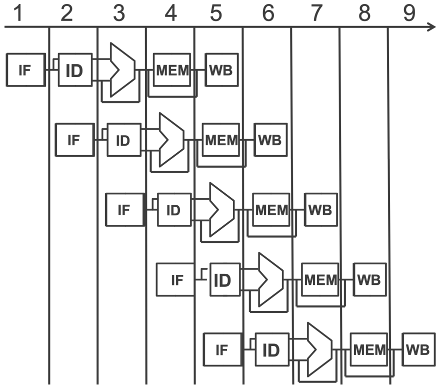


All the dependent actions are shown in color. The first instruction writes into x2, and all the following instructions read x2. This register is written in clock cycle 5, so the proper value is unavailable before clock cycle 5. The colored lines from the top datapath to the lower ones show the dependences. Those that must go backward in time are pipeline data hazards.

Practice

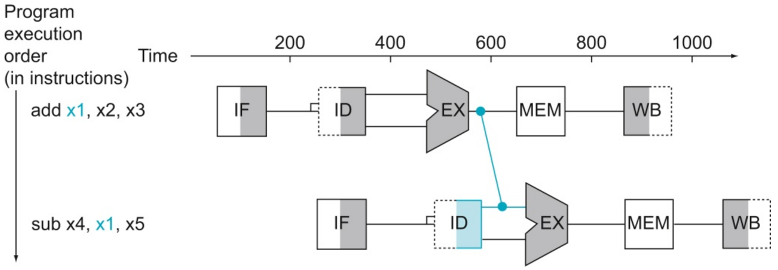
- Indicate all the data hazards?
  - draw lines between stages

```
add x3, x1, x2

sub x5, x3, x4

lw x6, 4(x3)

or x5, x3, x5

sw x6, 12(x3)
```



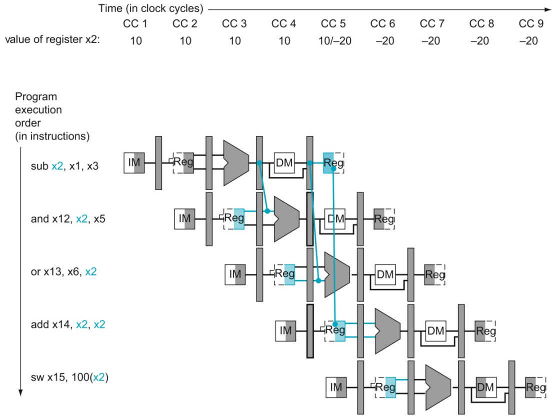
Forwarding (a.k.a. bypassing)

- Use result when it is computed
  - don't wait for it to be written to the register file
  - requires extra connections in the datapath



Forwarding the output of EX replaces the value from register x1 read in the second stage of sub

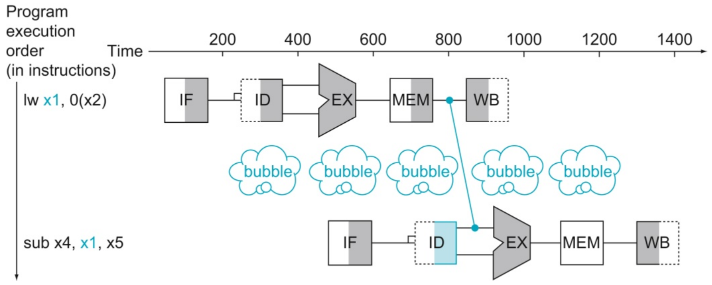
Forwarding



It is possible to supply the inputs to the ALU needed by the and instruction and or instruction by forwarding the results found in the pipeline registers. The values in the pipeline registers show that the desired value is available before it is written into the register file. We assume that the register file forwards values that are read and written during the same clock cycle, so the add does not stall, but the values come from the register file instead of a pipeline register. Register file "forwarding"—that is, the read gets the value of the write in that clock cycle—is why clock cycle 5 shows register x2 having the value 10 at the beginning and –20 at the end of the clock cycle.

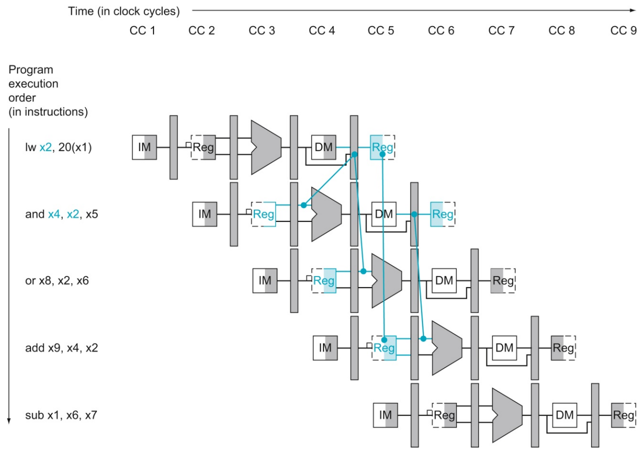
Load-use data hazard

- Can't always avoid stalls by forwarding
  - if value not computed when needed, can't forward "backward" in time



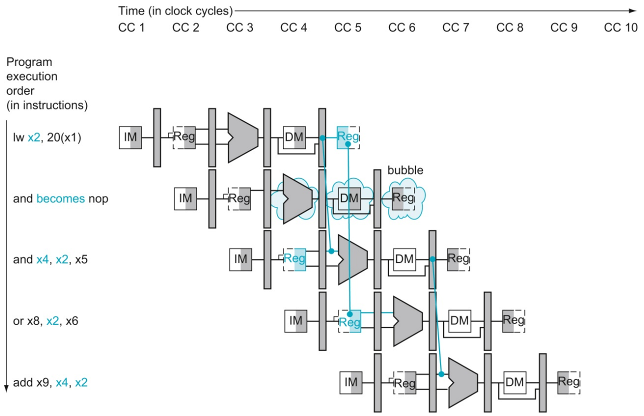
We need a stall even with forwarding when an instruction following a load tries to use the data. Without the stall, the path from memory access stage output to execution stage input would be going backward in time, which is impossible. This figure is actually a simplification, since we cannot know until after the subtract instruction is fetched and decoded whether or not a stall will be necessary.

Load-use data hazard



A pipelined sequence of instructions. Since the dependence between the load and the following instruction (and) goes backward in time, this hazard cannot be solved by forwarding. Hence, this combination must result in a stall by the hazard detection unit.

Inserting stalls



The way stalls are really inserted into the pipeline. A bubble is inserted beginning in clock cycle 4, by changing the and instruction to a nop. Note that the and instruction is really fetched and decoded in clock cycles 2 and 3, but its EX stage is delayed until clock cycle. Likewise, the or instruction is fetched in clock cycle 3, but its ID stage is delayed until clock cycle 5 (versus the unstalled clock cycle 4 position). After insertion of the bubble, all the dependences go forward in time and no further hazards occur.

Code scheduling to avoid stalls

- Reorder instructions

```
A = B + E;
C = B + F;
```

```
lw   x1, 0(x0)
lw   x2, 8(x0)
add  x3, x1, x2
sw   x3, 24(x0)
lw   x4, 16(x0)
add  x5, x1, x4
sw   x5, 32(x0)
```

Number of cycles?

```
lw   x1, 0(x0)
lw   x2, 8(x0)
lw   x4, 16(x0)
add  x3, x1, x2
sw   x3, 24(x0)
add  x5, x1, x4
sw   x5, 32(x0)
```

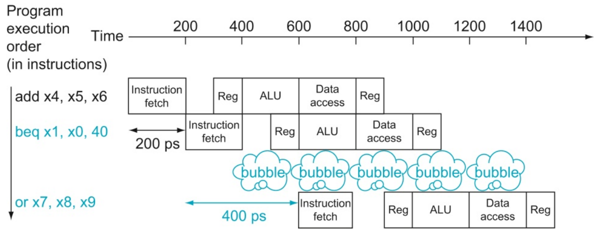
Number of cycles?

Control hazards

- Branch instructions impact control flow
  - fetching next instruction depends on branch outcome
  - pipeline can’t always fetch correct instruction
    - still working on ID stage of branch
- In RISC-V pipeline
  - need to compare registers and compute target early in the pipeline
  - **may add hardware to do it in ID stage**

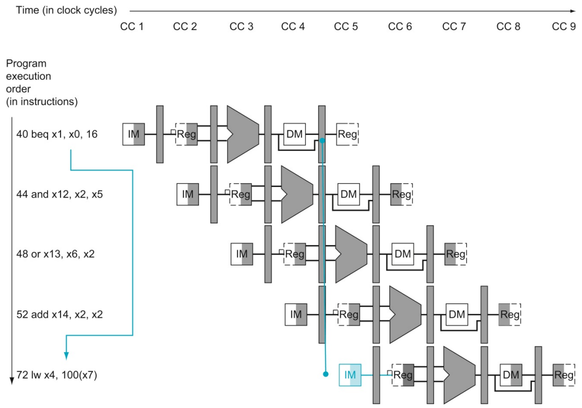
Stall on branch

- Wait until branch outcome determined before fetching next instruction
  - one cycle stall if branch condition is determined at ID
  - two cycles stall if branch condition is determined at EXE



Pipeline showing stalling on every conditional branch as solution to control hazards. This example assumes the conditional branch is taken, and the instruction at the destination of the branch is the or instruction. There is a one-stage pipeline stall, or bubble, after the branch.

Impact of the pipeline on branch instructions



The numbers to the left of the instruction (40, 44, …) are the addresses of the instructions. Since the branch instruction decides whether to branch in the MEM stage—clock cycle 4 for the beq instruction above—the three sequential instructions that follow the branch will be fetched and begin execution. Without intervention, those three following instructions will begin execution before beq branches to lw at location 72. (Figure on previous slide assumed extra hardware to reduce the control hazard to one clock cycle; this figure uses the nonoptimized datapath.)

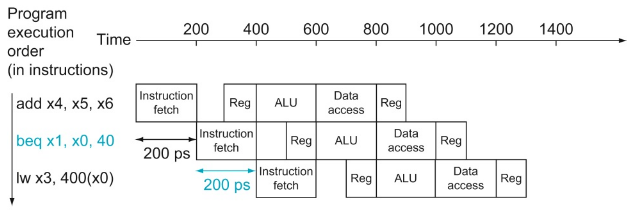
Branch prediction

Branch prediction

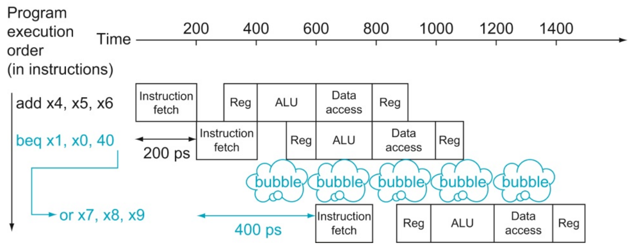
- Impact of pipeline depth
  - modern deep pipelines increase branch resolution latency
  - branch outcome determined several cycles after fetch
  - stalling until resolution significantly impacts performance
- Branch prediction strategies
  - **predict** branch outcome before resolution and continue fetching along predicted path
  - penalty occurs only on misprediction
    - required to flush the pipeline and to restart fetch from correct path
- RISC-V pipeline branch handling
  - **static prediction** assumes branches are not taken (fetch instruction after branch with no delay)
    - RISC-V ISA supports both static and dynamic prediction
  - hardware includes branch prediction buffers, branch target buffers, and branch history tables (needs recovery mechanisms for mispredictions)

Predicting branches are not taken

correct prediction



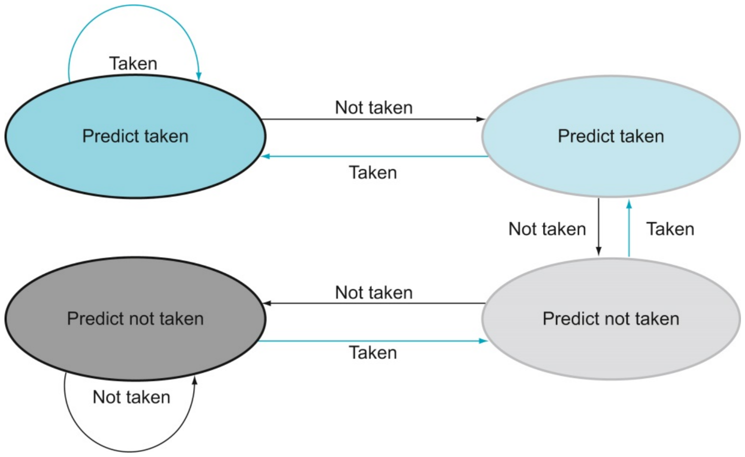
incorrect prediction



More realistic branch prediction

- Static branch prediction
  - based on typical branch behavior
  - example:
    - predict **backward** branches always taken (loops)
    - predict **forward** branches not taken (conditionals)
- Dynamic branch prediction
  - hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - assume future behavior will continue the trend
    - when wrong, stall while re-fetching, and update history

2-bit prediction

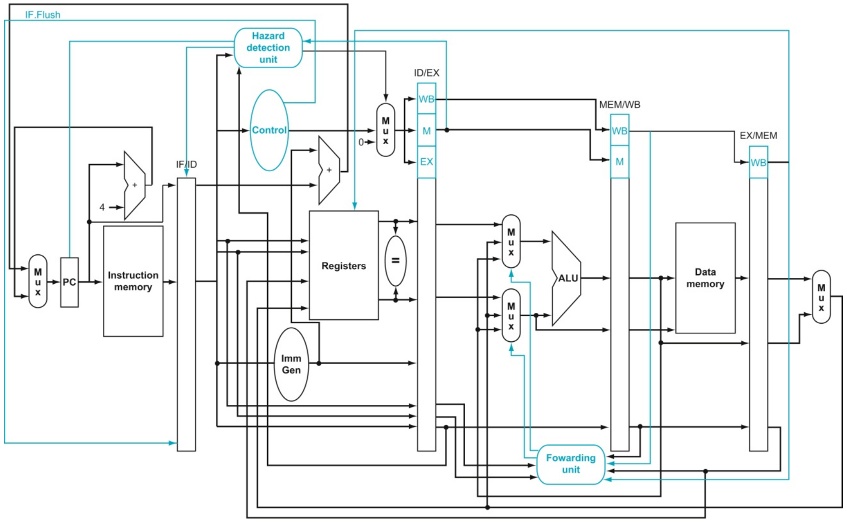


The 2 bits are used to encode the four states in the system

Pipeline summary

- Pipelining improves performance by increasing instruction throughput
  - executes multiple instructions in parallel
  - each instruction has the same latency
- Subject to hazards
  - structural, data, control
- Instruction set design affects complexity of pipeline implementation

Final datapath



Advanced techniques

Instruction-level parallelism (ILP)

- Basic pipelining
  - multiple instructions execute simultaneously in different stages
  - ideal CPI approaches 1
  - limited by pipeline hazards
- ILP techniques
  - deeper pipelines
    - divide execution into more stages
    - reduced work per stage → higher clock frequencies
    - trade-off: increased hazard penalties
  - multiple issue
    - multiple instructions issued per clock cycle
    - requires replicated functional units
    - performance, e.g., 4GHz 4-way multiple-issue
      - 16 BIPS, peak CPI = 0.25, peak IPC = 4 (dependencies/hazards reduce this in practice)

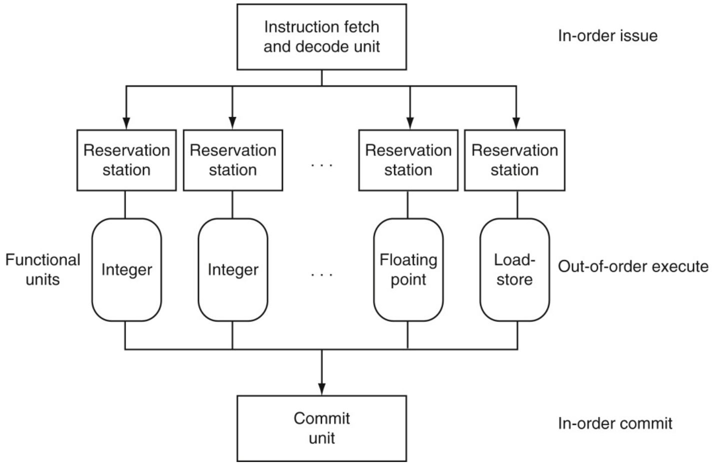
Multiple issue

- Static multiple issue
  - compiler groups instructions to be issued together
  - packages them into “issue slots”
  - compiler detects and avoids hazards
- Dynamic multiple issue
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

Speculation

- "Guess" what to do with an instruction
  - start operation as soon as possible
  - check whether guess was right
    - if so, complete the operation
    - if not, roll-back and do the right thing
- Common to static and dynamic multiple issue
  - examples
    - speculate on branch outcome
      - roll-back if path taken is different
    - speculate on load
      - roll-back if location is updated
  - predict and continue issuing, commit after outcome determined
    - use buffers

Dynamic pipeline scheduling

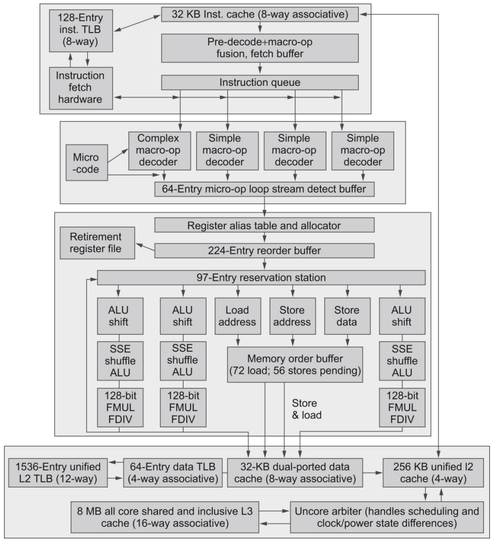


The three primary units of a dynamically scheduled pipeline. The final step of updating the state is also called retirement or graduation.

Intel microprocessors

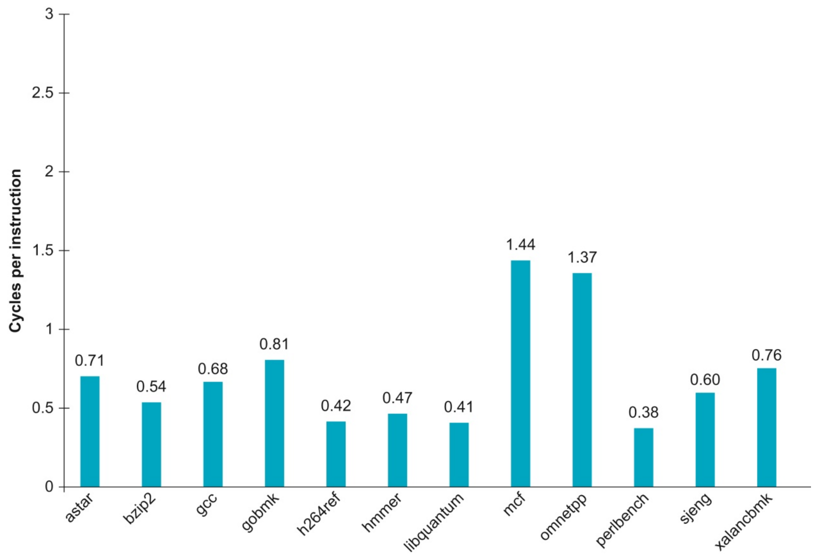
Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/ Speculation	Cores/ Chip	Power
Intel 486	1989	25 MHz	5	1	No	1	5W
Intel Pentium	1993	66 MHz	5	2	No	1	10W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103W
Intel Core	2006	3000 MHz	14	4	Yes	2	75W
Intel Core i7 Nehalem	2008	3600 MHz	14	4	Yes	2-4	87W
Intel Core Westmere	2010	3730 MHz	14	4	Yes	6	130W
Intel Core i7 Ivy Bridge	2012	3400 MHz	14	4	Yes	6	130W
Intel Core Broadwell	2014	3700 MHz	14	4	Yes	10	140W
Intel Core i9 Skylake	2016	3100 MHz	14	4	Yes	14	165W
Intel Ice Lake	2018	4200 MHz	14	4	Yes	16	185W

Intel Core i7 pipeline

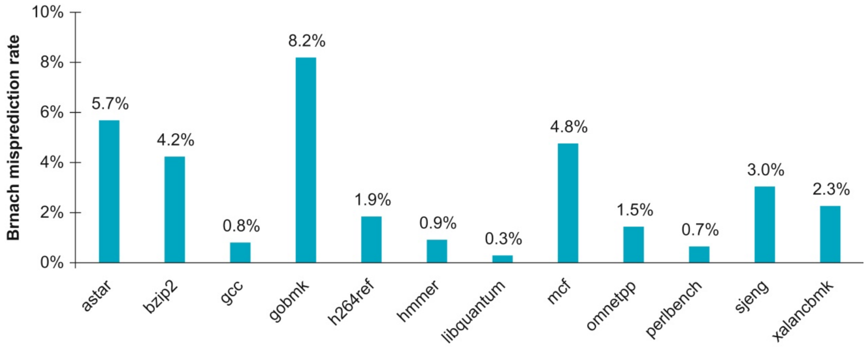

128-Entry inst. TLB (8-way)
32 KB Inst. cache (8-way associative)
Pre-decode+macro-op fusion, fetch buffer
Instruction fetch hardware
Instruction queue
Micro -code
Complex macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
64-Entry micro-op loop stream detect buffer
Register alias table and allocator
Retirement register file
224-Entry reorder buffer
97-Entry reservation station
ALU shift
ALU shift
Load address
Store address
Store data
ALU shift
SSE shuffle ALU
SSE shuffle ALU
Memory order buffer (72 load; 56 stores pending)
SSE shuffle ALU
128-bit FMUL FDIV
128-bit FMUL FDIV
Store & load
128-bit FMUL FDIV
1536-Entry unified L2 TLB (12-way)
64-Entry data TLB (4-way associative)
32-KB dual-ported data cache (8-way associative)
256 KB unified l2 cache (4-way)
8 MB all core shared and inclusive L3 cache (16-way associative)
Uncore arbiter (handles scheduling and clock/power state differences)

The total pipeline depth is 14 stages, with branch mispredictions typically costing 17 cycles. This design can buffer 72 loads and 56 stores. The six independent functional units (6-wide superscalar) can each begin execution of a ready micro-operation in the same cycle.

CPI for the i7 6700 (SPECCPUint2006)



Misprediction rate for the i7 6700 (SPECCPUint2006)


Brnach misprediction rate
astar
bzip2
gcc
gobmk
h264ref
hmmer
libquantum
mcf
omnetpp
perlbench
sjeng
xalancbmk