CS 423
Computer Architecture
Spring 2012

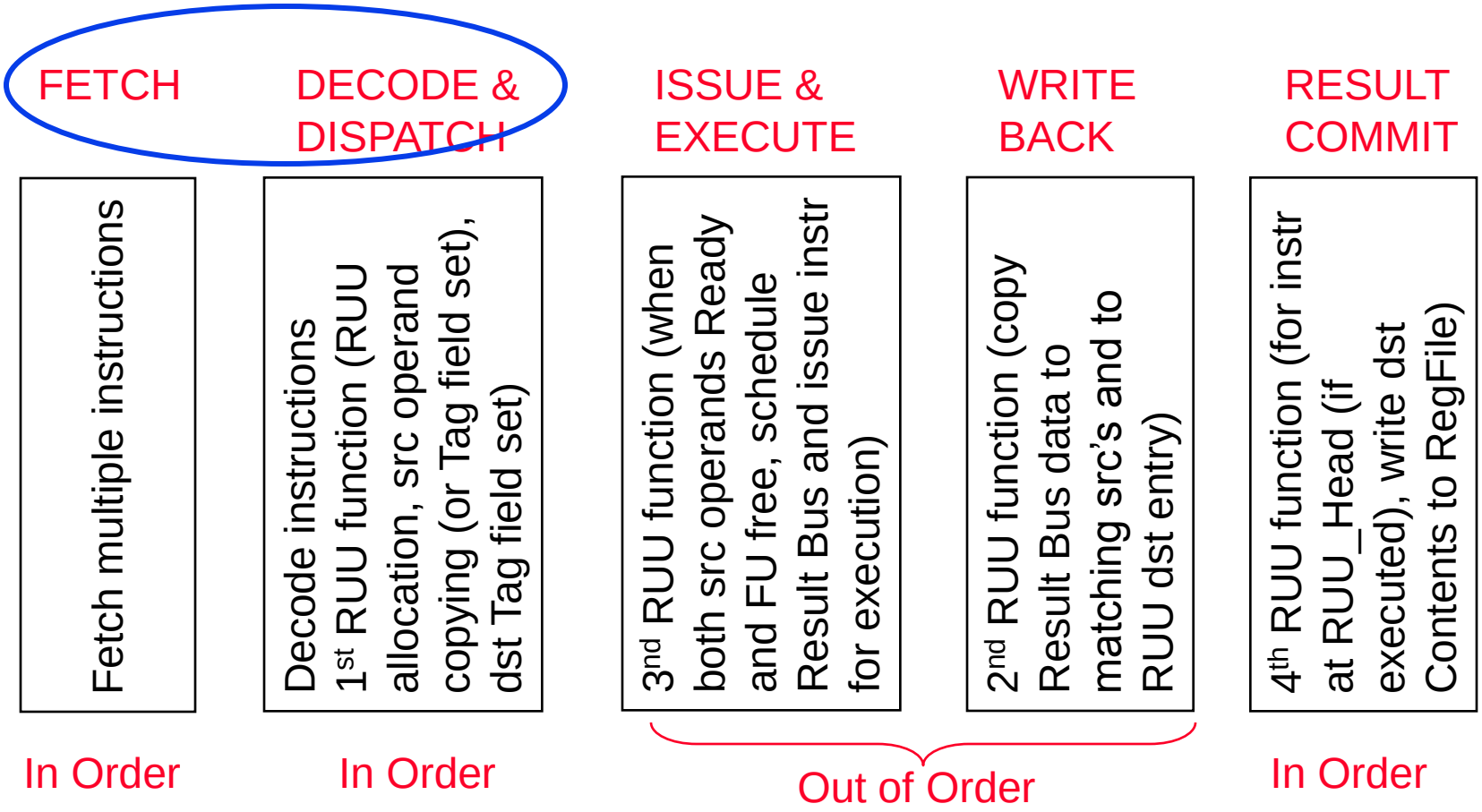
Lecture 05:
SS Front End / Back End Issues

Ozcan Ozturk

http://www.cs.bilkent.edu.tr/~ozturk/cs423/

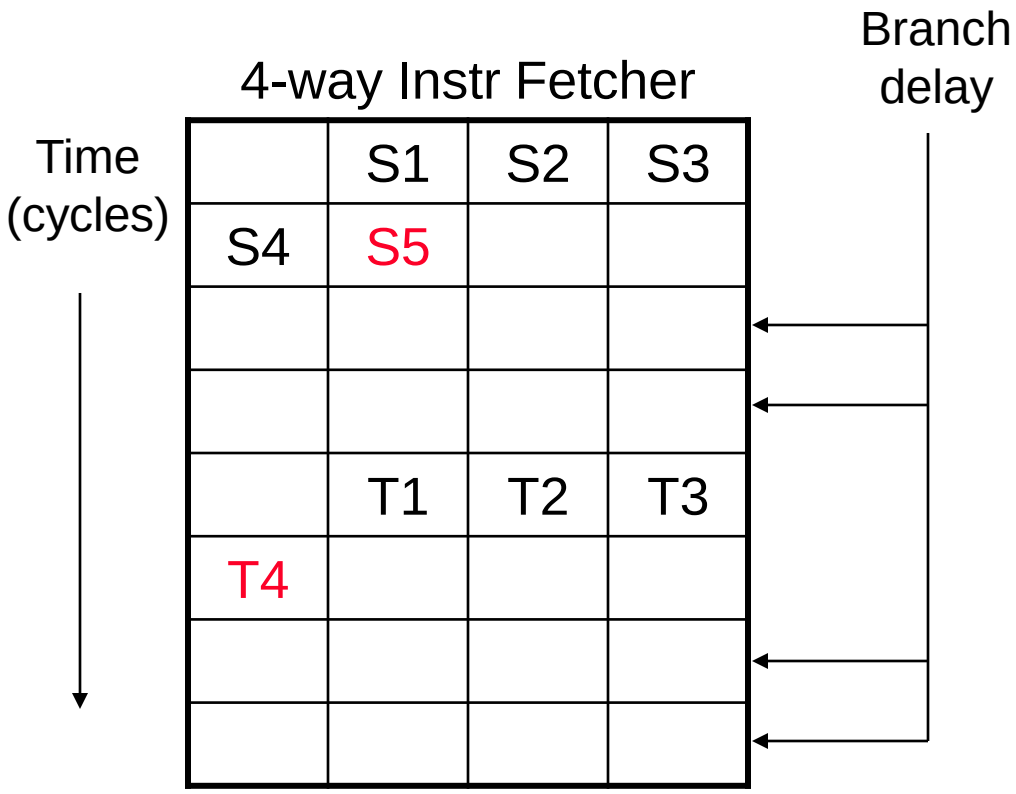
[Adapted from *Computer Organization and Design*,
Patterson & Hennessy, © 2005, UCB]

SS Pipeline

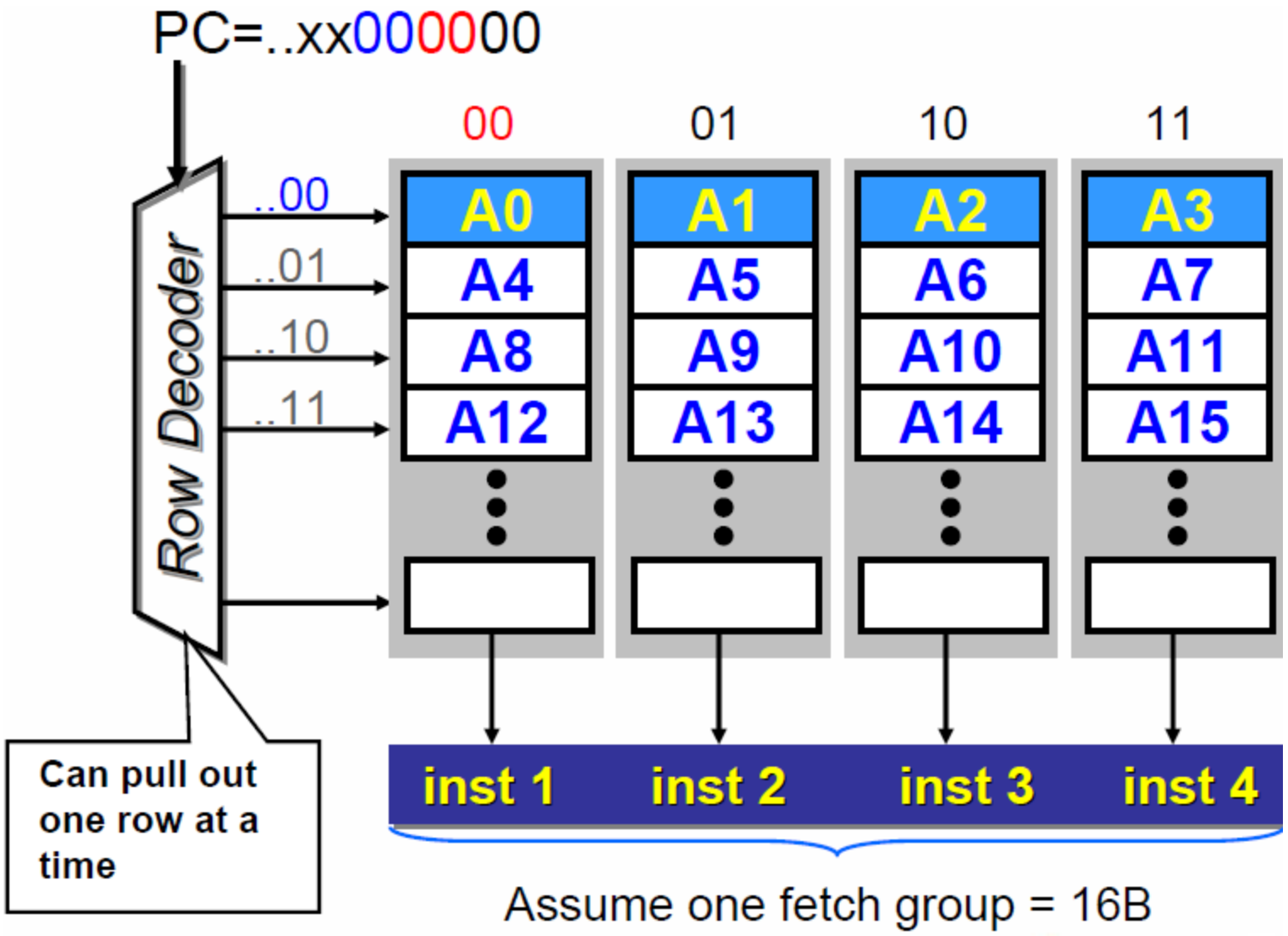

FETCH
DECODE & DISPATCH
ISSUE & EXECUTE
WRITE BACK
RESULT COMMIT
Fetch multiple instructions
Decode instructions
1st RUU function (RUU allocation, src operand copying (or Tag field set), dst Tag field set)
3nd RUU function (when both src operands Ready and FU free, schedule Result Bus and issue instr for execution)
2nd RUU function (copy Result Bus data to matching src's and to RUU dst entry)
4th RUU function (for instr at RUU_Head (if executed), write dst Contents to RegFile)
In Order
In Order
Out of Order
In Order

Instruction Fetch Sequences

- Instruction run – number of instructions (run length) fetched between taken branches
  - Instruction fetcher operates most efficiently when processing long runs – unfortunately runs are usually quite short
- The average run length is about six instructions
- Instruction bandwidth of only 1.125 instructions per cycle
  - 9 instructions in 8 cycles



Instruction Fetch Misalignment



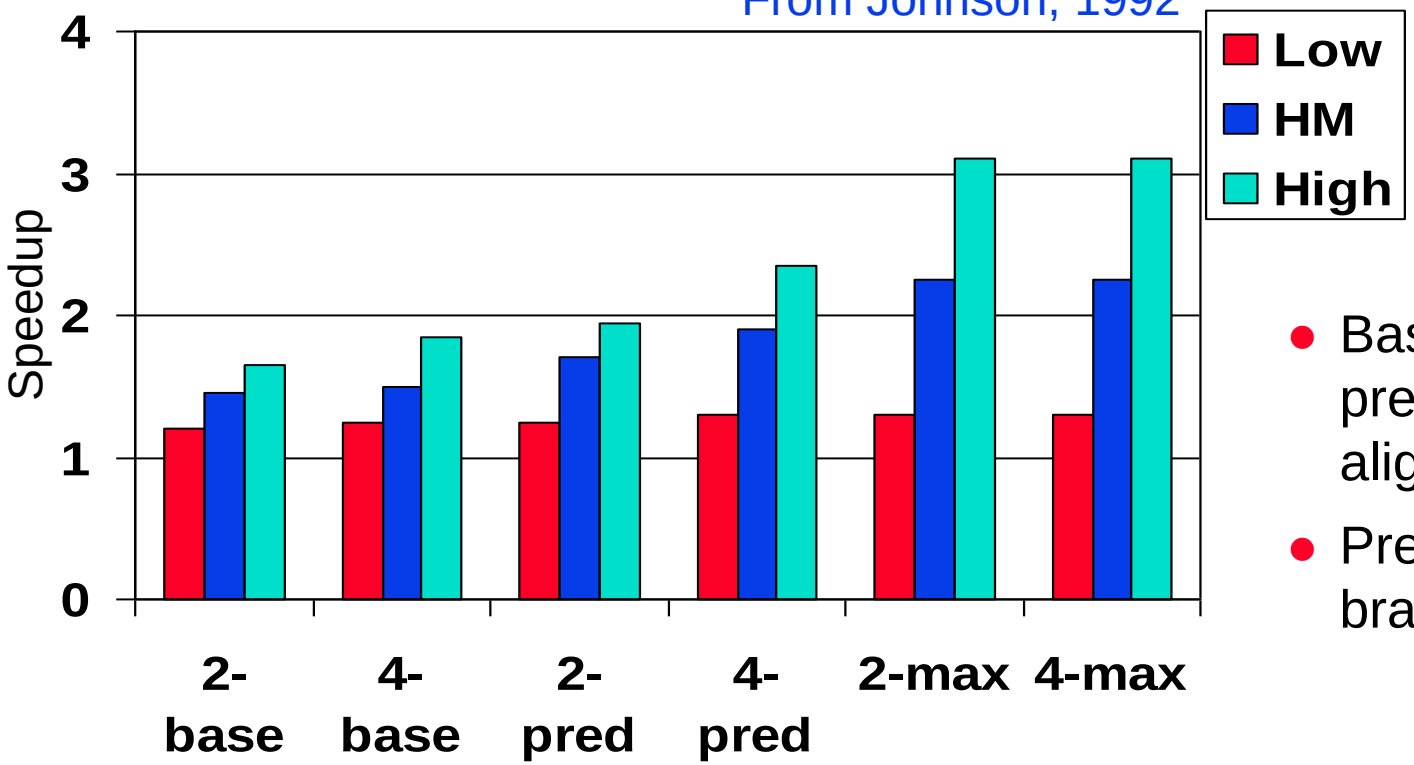
Instruction Fetch Inefficiencies

- Fetcher can't provide adequate bandwidth to the decoder to exploit the available ILP because
  - Decoder is idle while the outcome of the branch is determined
    - Can (mostly) fix with dynamic branch prediction
  - Instruction fetch misalignment prevents the decoder from operating at full capacity even when the decoder *is* processing valid instructions
    - The fetcher can align fetched instructions to avoid wasted decoder slots
    - If supported by dynamic branch prediction, the fetcher can also merge instructions from different runs

- Aligning and merging can only be done if the fetcher has the sufficient bandwidth (i.e., the fetch rate is faster than the decode rate)

Speedups of Fetch Alternatives

From Johnson, 1992

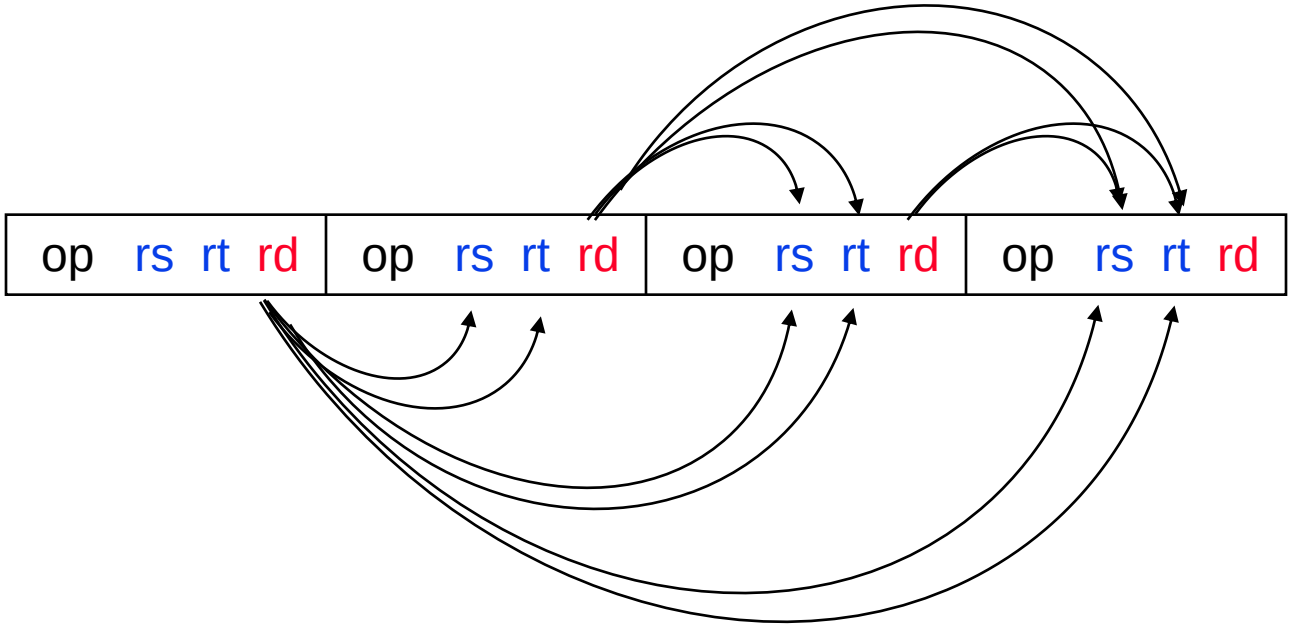


- Base: no prediction and no alignment
- Pred: dynamic branch prediction

- A 4-way instr fetcher out performs a 2-way instr fetcher
  - It has twice the potential instruction bandwidth
  - But it requires twice as much decoder hardware to keep up (e.g., in decoders and in ports and buses)

4-Way Decoder Implementation

- A 4-way instr fetcher has higher fetch bandwidth but at what cost ?
  - 12 dependency checks between the 4 decode instruction



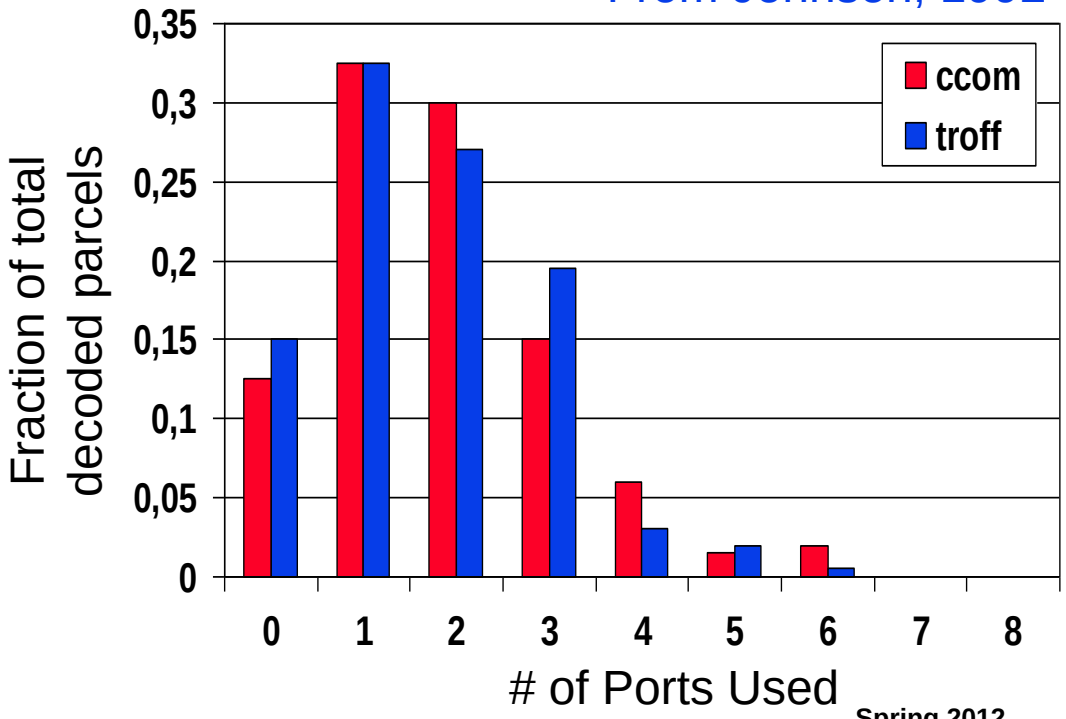
  - 8 read ports on the RegFile and 8 write ports to the RUU and 8 buses to distribute those source operands (or their RegFile addr || LI)

Reducing 4-Way Decoder Hardware

- Limiting the number of RegFile read ports, buses and RUU write ports is acceptable since
  - Not all decoded instructions access two registers
  - Not all decoded instruction are valid (because of misalignment)
  - Some decoded instructions have dependences on one or more simultaneously decoded instructions

- The register demand means a 8 port capacity would be wasted
- 4 ports reduces average performance < 2%

From Johnson, 1992



Arbitrating Read Port and Bus Usage

- If only 4 read ports and 4 buses are provided, have to determine which instr's get first access to them
  - If more than 4 ports are needed to dispatch the decoded instructions, then instruction fetch and decode must stall

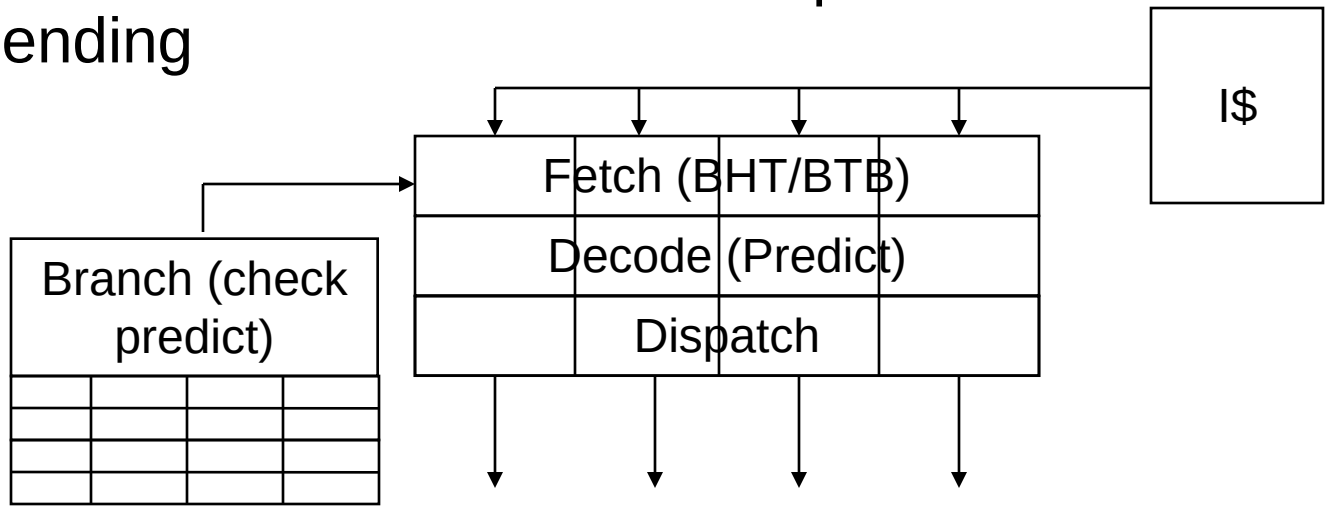
- Prioritized register identifier selection for port usage must be accomplished within about half a processor cycle
  - If the first decoder position instr (i.e., the first instr in program order) requires register access, it is always enabled on the first and second, if it has two source operands, ports
  - Such arbitration continues in sequence for the second, third, and fourth decoder position instr's

SS Branch Prediction

- Recall that for a branch prediction in a scalar pipeline we needed
  - A mechanism to predict the branch outcome: a BHT (branch history table) in the fetch stage
  - A way to fetch two instructions – the sequential instruction (I$) *and* the branch target instruction (BTB (branch target buffer))
  - A way to ensure that instructions active in the pipeline following the branch didn't change the machine state until the branch outcome was known
    - Allowed to complete (in order commit) on correct prediction
    - Flushed on mispredict and restart
- With a SS machine, it is possible to have many such instructions after predicted branches *active* in the pipeline
  - Flag instructions following branches as speculative until the branch outcome is known

Implementing Branches

- A SS processor could have more than one branch per fetch set and could have several uncompleted branches pending at any time



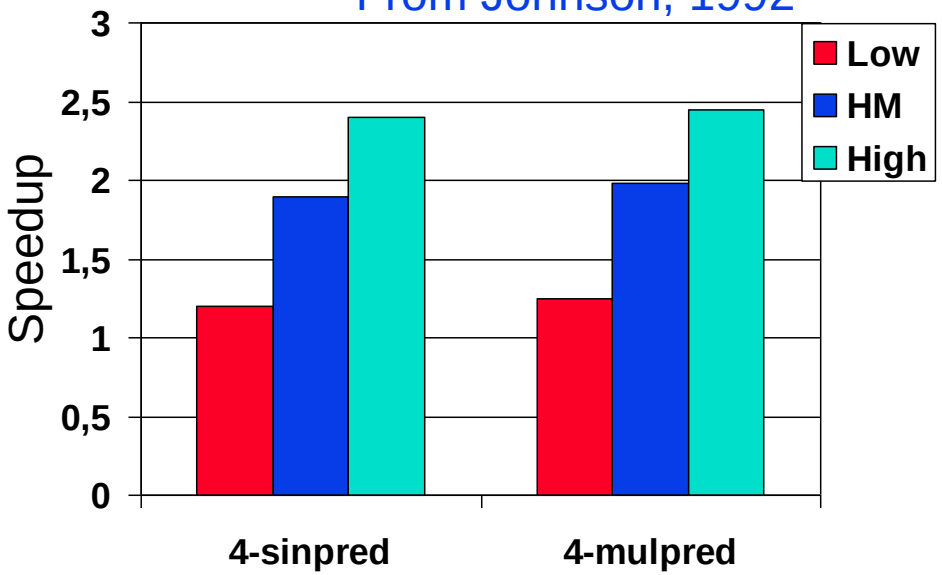
- Must access BHT/BTB for all branch instr's in the fetch set during fetch to reduced branch delay (i.e., need a 4 read-port BHT/BTB for 4-instr fetcher)
  - Pass BHT information to decode stage
  - After decode, choose between I$ set and BTB sets to determine the next fetch set

Decoding & Dispatching Branches

- While multiple branches could be dispatched per cycle, incur only a slight performance decrease (about 2%) from imposing a decoder limit of one branch per fetch set since typically only one branch per cycle can be executed (usually only have one branch FU)

- Having minimum branch delay is more important that decoding multiple branches per cycle

From Johnson, 1992

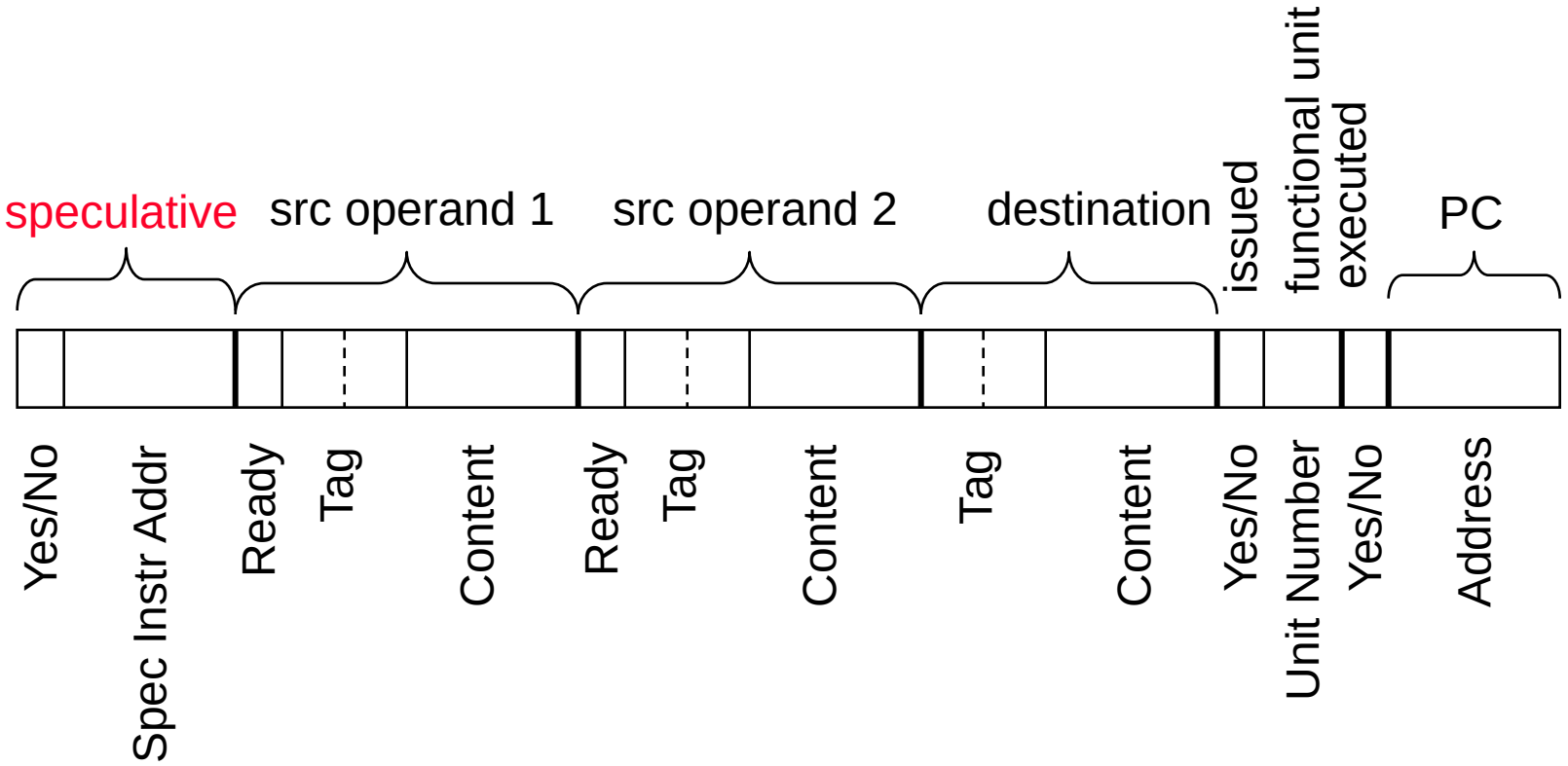


Speculative Instructions

- Speculation – The processor (or compiler) guesses the outcome of an instruction (e.g., branches, loads) so as to enable execution of other instructions that depend on the speculated instruction
  - One of the most important methods for finding more ILP in SS and VLIW processors
- Producing correct results requires result checking, recovery and restart hardware mechanisms
  - Checking mechanisms to see if the prediction was correct
  - Recovery mechanisms to cancel the effects of instructions that were issued under false assumptions (e.g., branch misprediction)
  - Restart mechanisms to reestablish the correct instruction sequence
    - For branches the correct program counter restart value is known when the branch outcome is determined

RUU Speculation Field Support

- For dependent speculative instr's, the speculative flag is set to Yes until the outcome of the driving instr (i.e., the branch) is determined. Then an associate comparison of that branch's PC addr and the RUU's SIA fields can be done.



Branch Execution

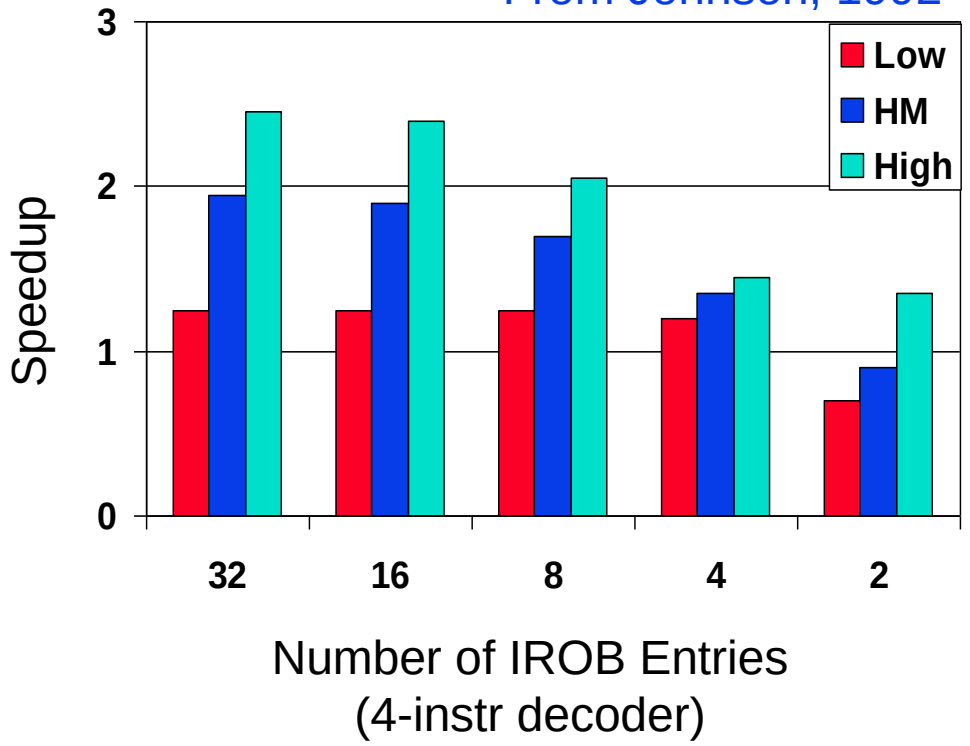
- If the branch was not mispredicted, then the branch and its trailing instructions can commit when at RUU_Head
- If the branch was mispredicted, then all subsequent instr's must be discarded (even though subsequent *branches* may have been correctly predicted)
- When there is an exception, all of the RUU entries are discarded in a *single cycle* and instruction stream fetching restarts on the next cycle. Thus, the RUU provides an easy way to discard instructions coming after a mispredicted branch.

Effects of RUU Size on Performance

- Since instruction decoding must stall when there is no free RUU entry, the RUU should be large enough to accept *all* instructions during the expected dispatch-to-commit time period

- Performance decreases markedly with 8 and 4 entries
- For 2 entries the performance is worse than the scalar processor
  - Why?

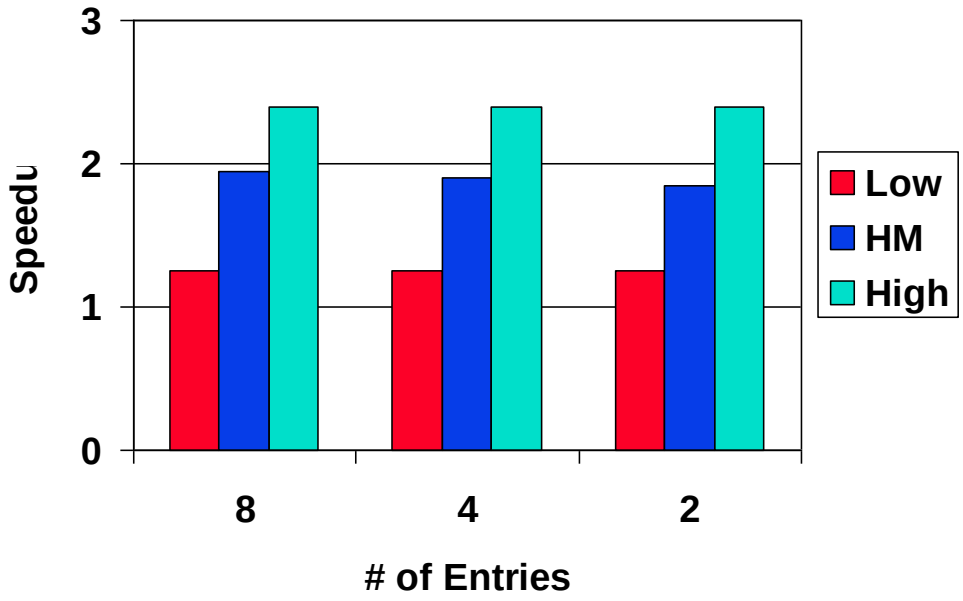
From Johnson, 1992



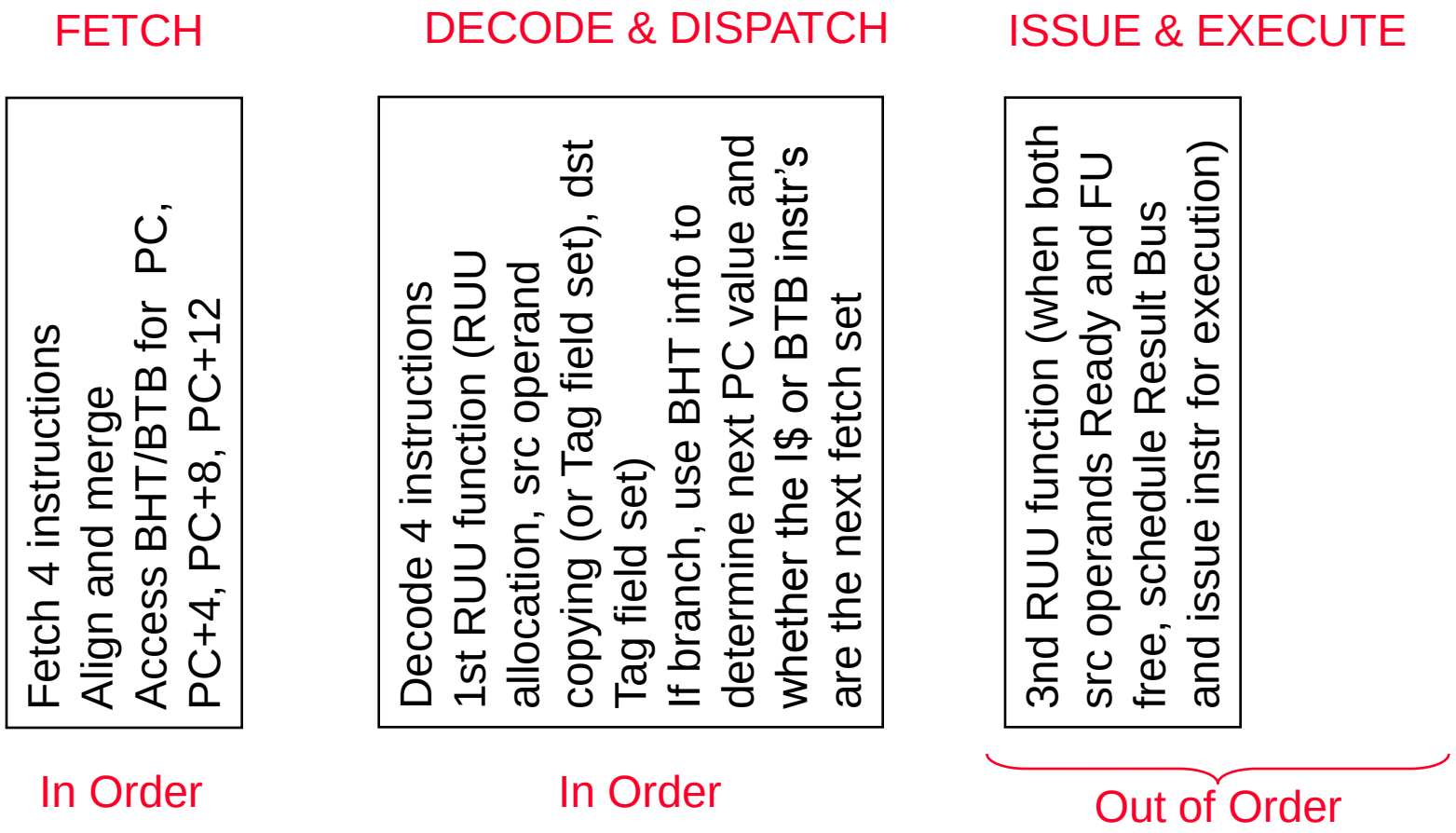
Effects of LSQ Size on Performance

- Since instruction decoding must stall when there is no free LSQ entry, the LSQ should be large enough but
- the LSQ size has relatively little impact on performance
  - With a 4-way decoder, a 4-entry LSQ only incurs a 1% speedup loss over an 8-entry LSQ
  - Smaller LSQ facilitate dependency checking

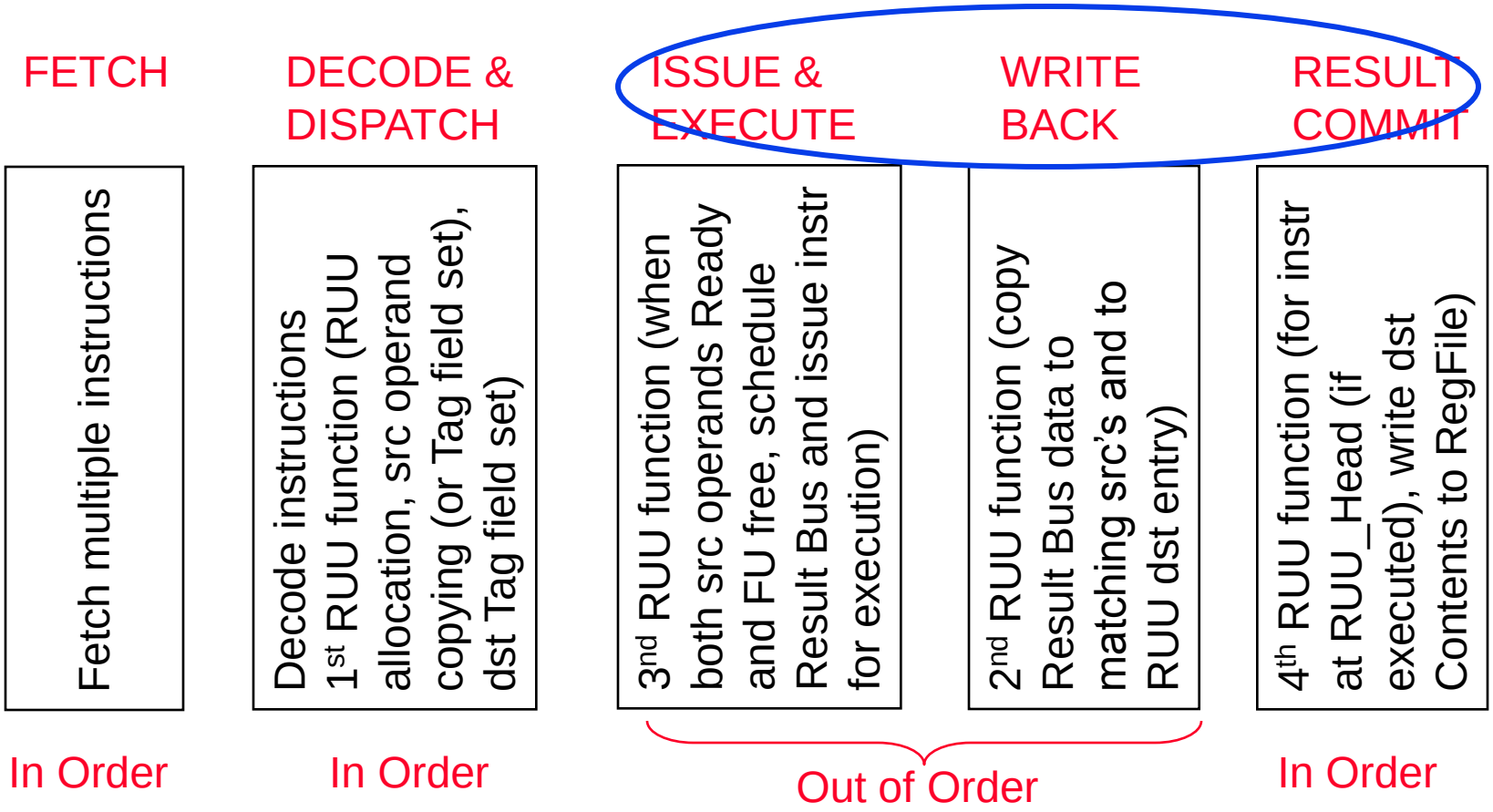
From Johnson, 1992



SS Fetch and Decode Pipeline Stages



SS Pipeline

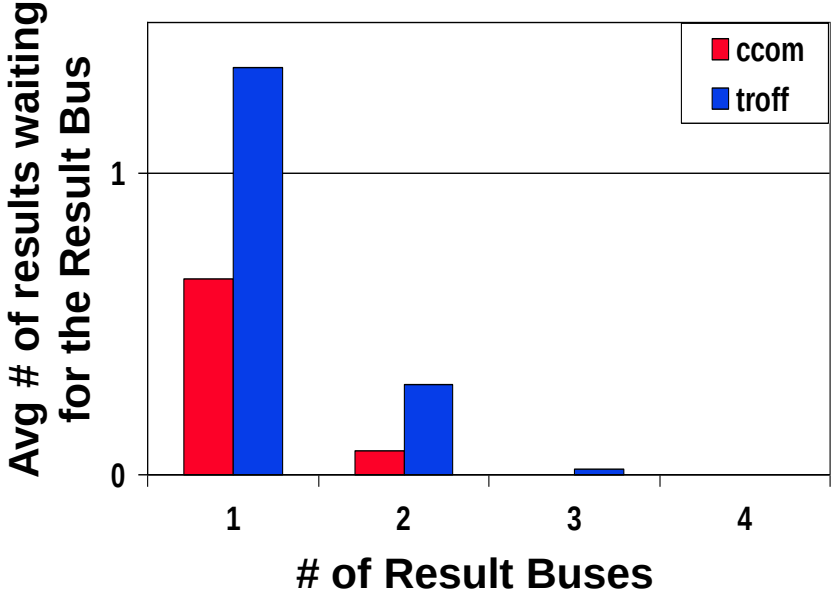

FETCH
DECODE & DISPATCH
ISSUE & EXECUTE
WRITE BACK
RESULT COMMIT
Fetch multiple instructions
Decode instructions 1st RUU function (RUU allocation, src operand copying (or Tag field set), dst Tag field set)
3nd RUU function (when both src operands Ready and FU free, schedule Result Bus and issue instr for execution)
2nd RUU function (copy Result Bus data to matching src's and to RUU dst entry)
4th RUU function (for instr at RUU_Head (if executed), write dst Contents to RegFile)
In Order
In Order
Out of Order
In Order

Result Buses Utilization

- Our SS model has only one Result Bus to carry results generated by the FU's to the RUU and LSQ
  - Even at the high levels of performance, the utilization of the Result Bus is only about 70% (i.e., the fraction of capacity actually used)

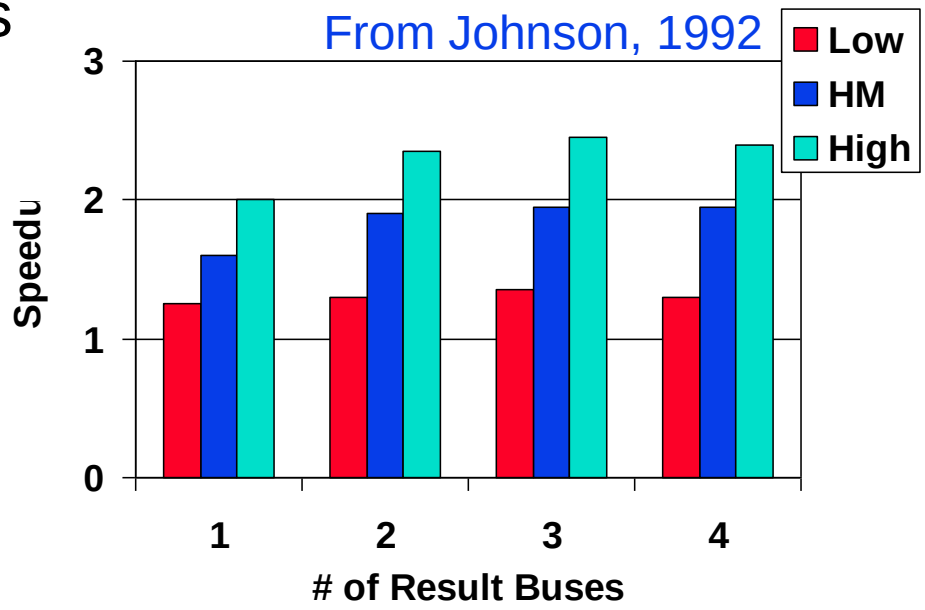
- If a FU requests for the Result Bus is not granted, instruction issue to that FU is stalled until the bus request can be granted (i.e., the FU remains "busy")

From Johnson, 1992



Arbitrating For Result Buses

- Reducing the impact of bus contention by adding a second Result Bus improves performance (by almost 19%)
- But adding a third Results Buses yields only a very small improvement in performance (less than 3%)



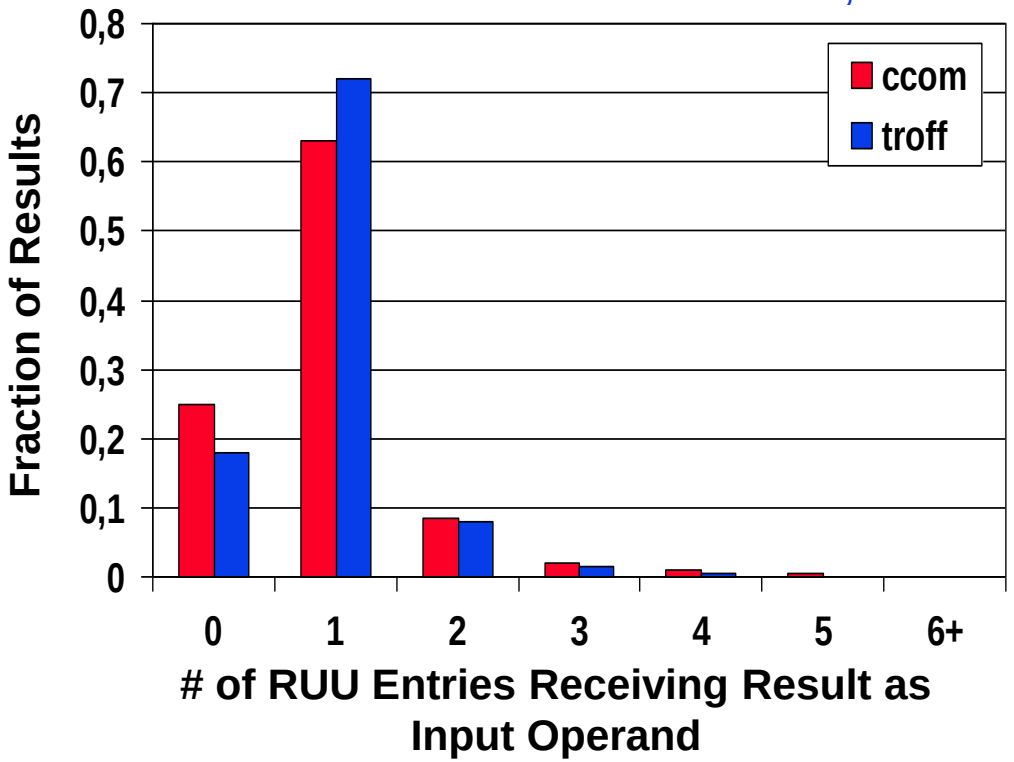
- And since there are usually fewer Result Buses than FUs, the FUs must continue to arbitrate for use of the existing Result Buses
  - The arbiter not only decides which FU is granted use of the Result Buses, but also which of the two buses is to be used
    - Prioritizing old requests over new helps prevent starvation

Result Forwarding

- Result forwarding supplies operands directly to the waiting instr's in the RUU to resolve true dependencies that could not be resolved during decode
  - The cost of forwarding is the comparison logic in the RUU to compare the Result Bus Tag to the source operand Tags
  - Need a set of comparators for each Result Bus

- About 2/3rd of all results are forwarded to one waiting operand, and about 1/6th are forwarded to more than one

From Johnson, 1992



Performance Advantages

- Hardware complexity arises from four major hardware features
  - Out-of-order issue
  - Register renaming
  - Branch prediction
  - 4-way instruction fetch and decode

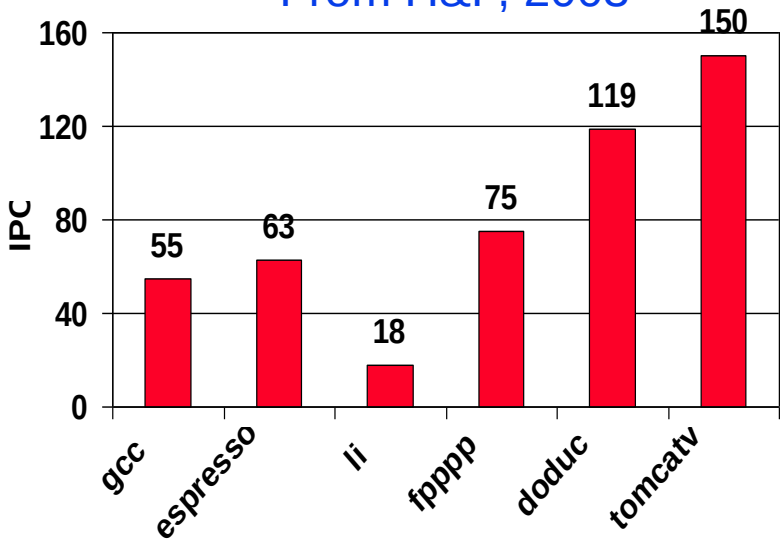
From Johnson, 1992

OOI	Register Renaming	Branch Prediction	4-way Fetch & Decode
52%	36%	30%	18%

ILP in a Perfect OOI-OOC Processor

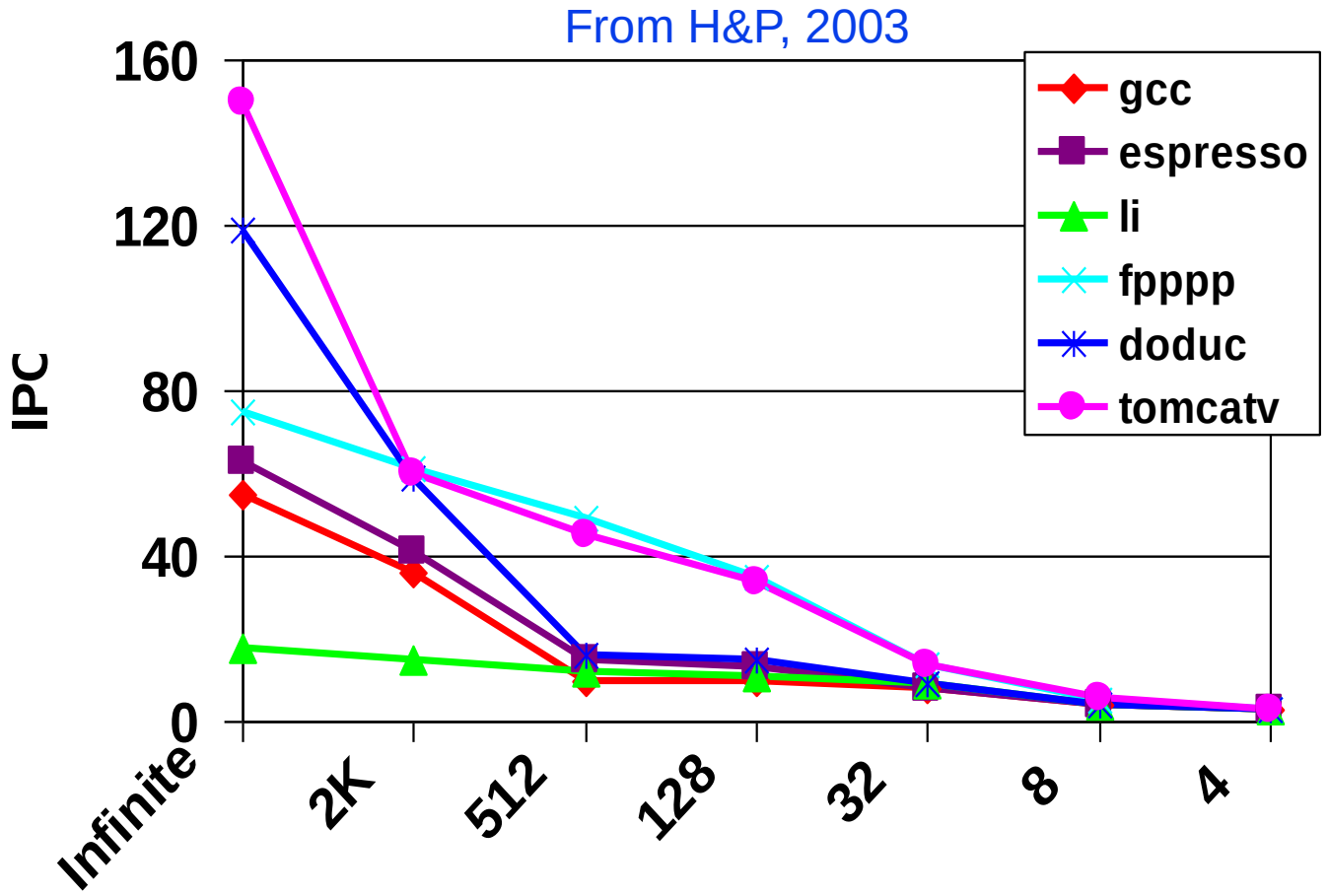
- The perfect processor has
  - An infinite number of rename registers that eliminates all storage hazards (i.e., write-before-write and write-before-read)
  - No (fetch, decode, dispatch, issue, FU, buses, ports) limit on the number of instr's that can begin execution simultaneously as long as read-before-write true data hazards are not present
  - Perfect branch and jump (including jump register) prediction
  - Loads can be moved before stores (as long as the addresses are not identical) with memory address analysis
  - All FU's have a 1 cycle latency
  - Perfect caches with 1 cycle latency

From H&P, 2003



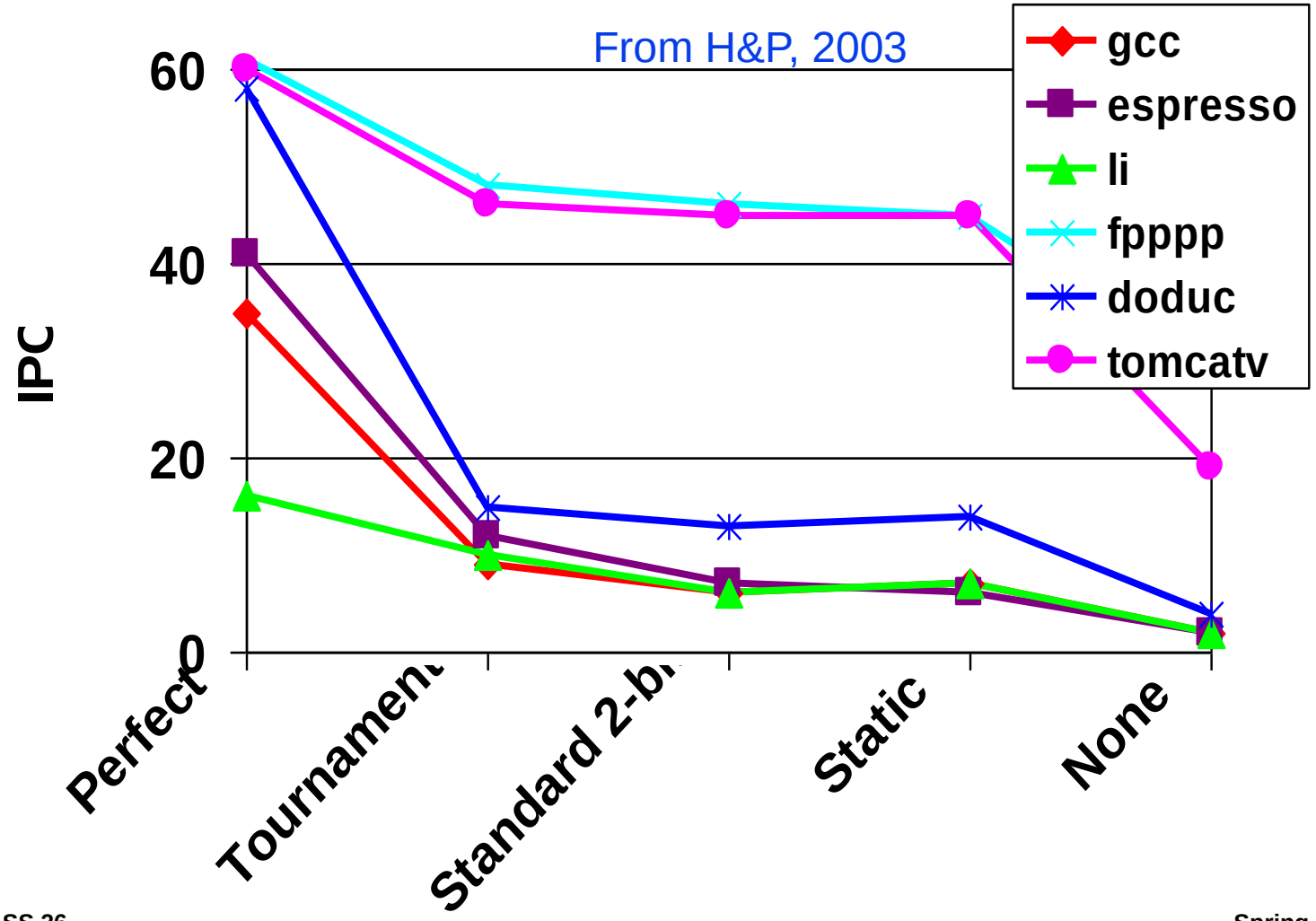
Effect of Instruction Window Size on ILP

- Instruction window – the set of instructions that are examined simultaneously for execution



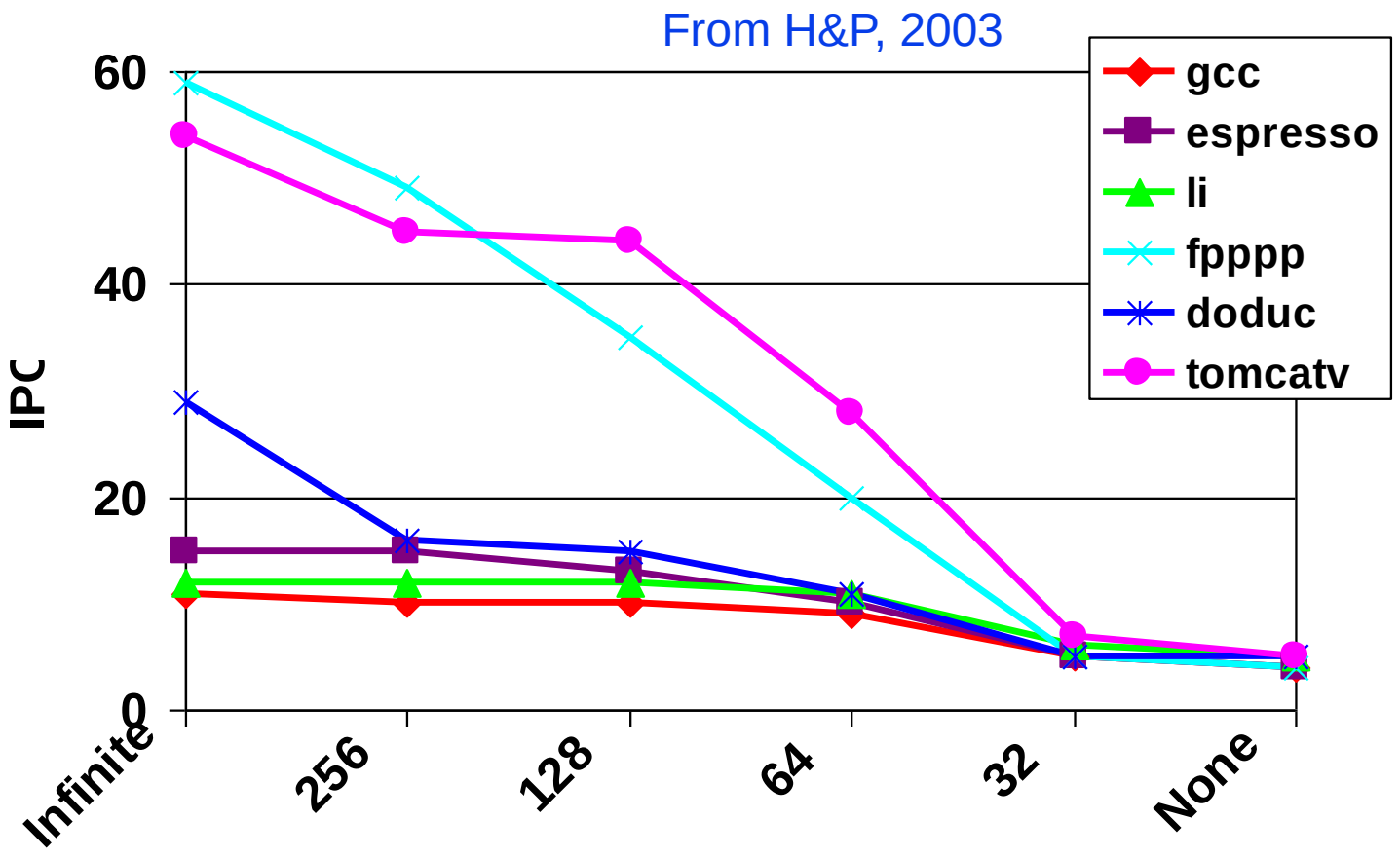
Effect of Realistic Branch Prediction on ILP

- On a processor with an instruction window size of 2K and maximum 64-way issue capability



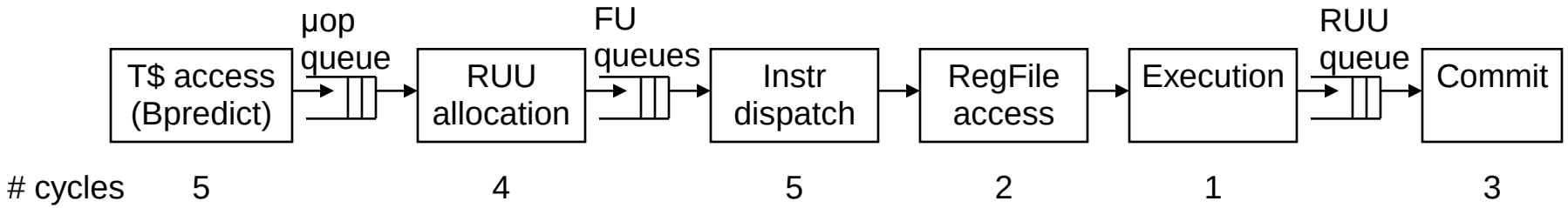
Effect of Finite Rename Registers

- On a processor with an instruction window size of 2K, maximum 64-way issue capability, and a tournament branch predictor with 8K entries



A SS Example

- Intel Pentium 4 (IA-32 ISA)
  - Decodes the IA-32 instructions into microoperations
  - Does register renaming with a RUU-like structure
  - Has a 20 stage pipeline



  - 7 FUs: 2 integer ALUs, 1 FP ALU, 1FP move, load, store, complex
  - Up to 126 instructions in flight, including 48 loads and 24 stores
  - 4K entry branch predictor