Modern Processors

RISC Architectures

Figures used from:
Manolis Katevenis, ”RISC Architectures”, Ch. 20 in
Zomaya, A.Y.H. (ed), *Parallel and Distributed Computing Handbook*,
McGraw-Hill, 1996

RISC Characteristics

- Simplified instruction set
- Register-to-register arithmetic instructions
- Memory accessed only by load and store instructions
- A single, simple addressing mode
- Fixed sized instructions
- A few, simple instruction formats
- Pipelining
- Optimizing compilers
- Small and simple circuits for high clock rate and reduced design time, design cost, and silicon area

Architectural Characteristics of RISC

- Register orientation
  - register file is small, thus fast
  - register file can have multiport access
  - short address, no effective-address calculation
  - optimizing compiler to make best use of registers
- Load/store architecture
  - information *transfer* separated from *operation*
- Fixed instruction size
  - simplifies implementation, but gives less compact code
- Fixed position of source operands in instruction format
- Single result per instruction
- Regularity
  - Operations are similar: can all be handled by the same subsystems
- Provide primitives - not solutions
  - Instruction set stays fixed for years
  - Language-oriented special features left for compiler
  - Compiler optimizes - not human assembly programmer

Pipelining and Bypassing

- The basic 5-stage RISC pipeline
  - Fig. 20.1, Fig. 20.2
- Dependencies and bypassing
  - Example, Fig. 20.3
  - More complicated example, Fig. 20.4

Basic RISC Pipeline

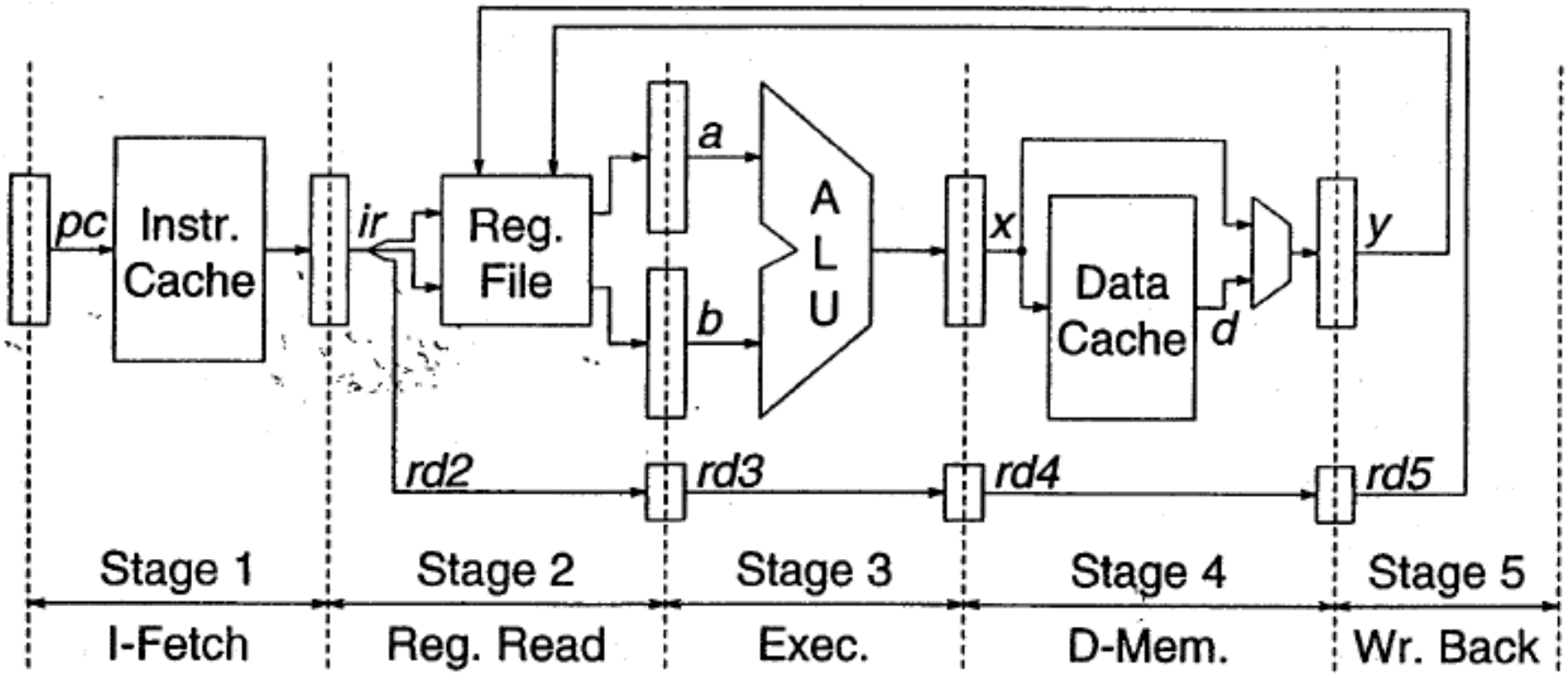

Figure 20.1 Simplified data path of a five-stage RISC pipeline

Pipelined Instruction Execution

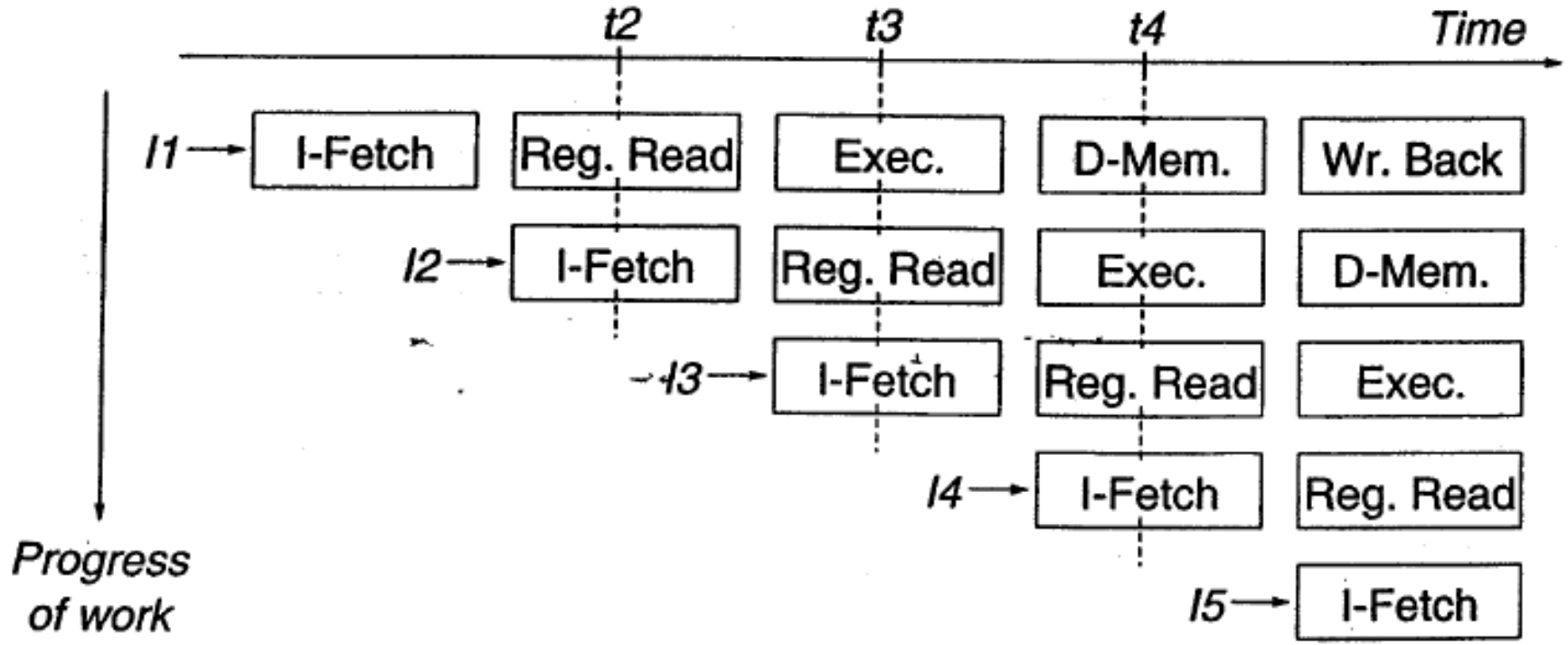

Figure 20.2 The basic five-stage RISC pipeline

Bypassing to enable dense register operations

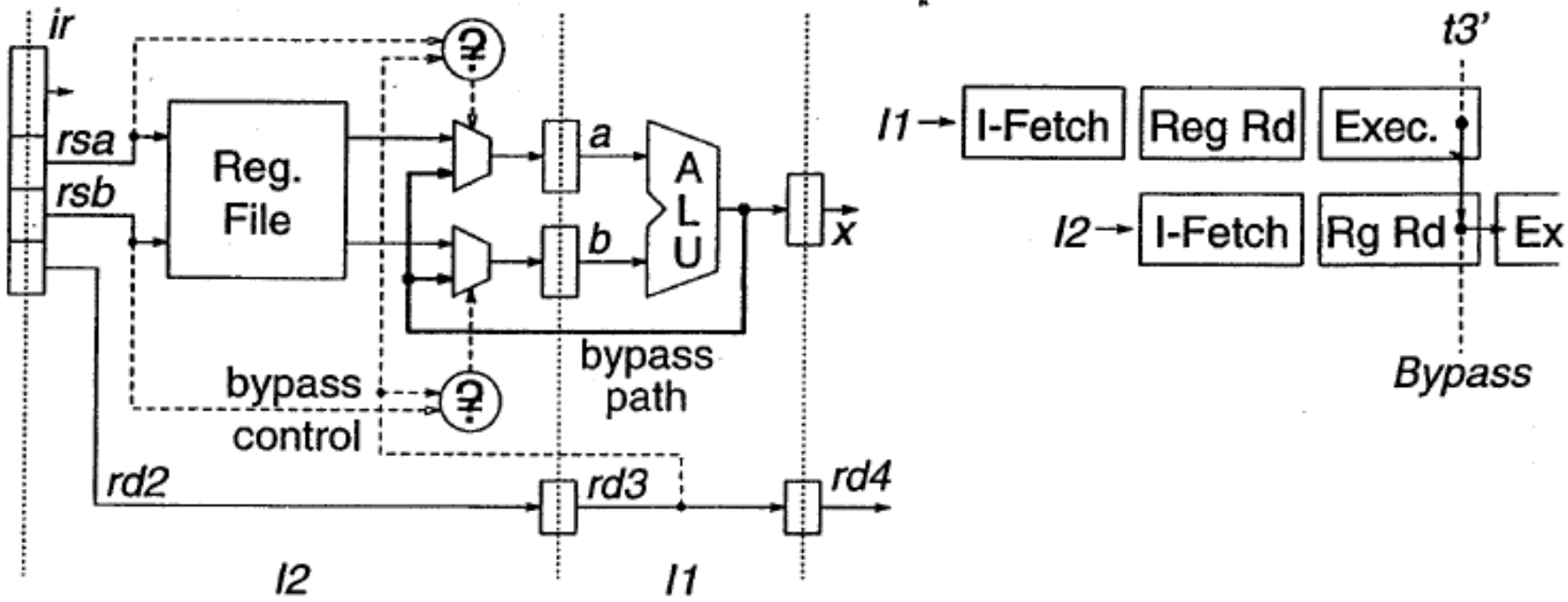

Figure 20.3 Bypassing from the third to the second stage

Bypassing in load instructions

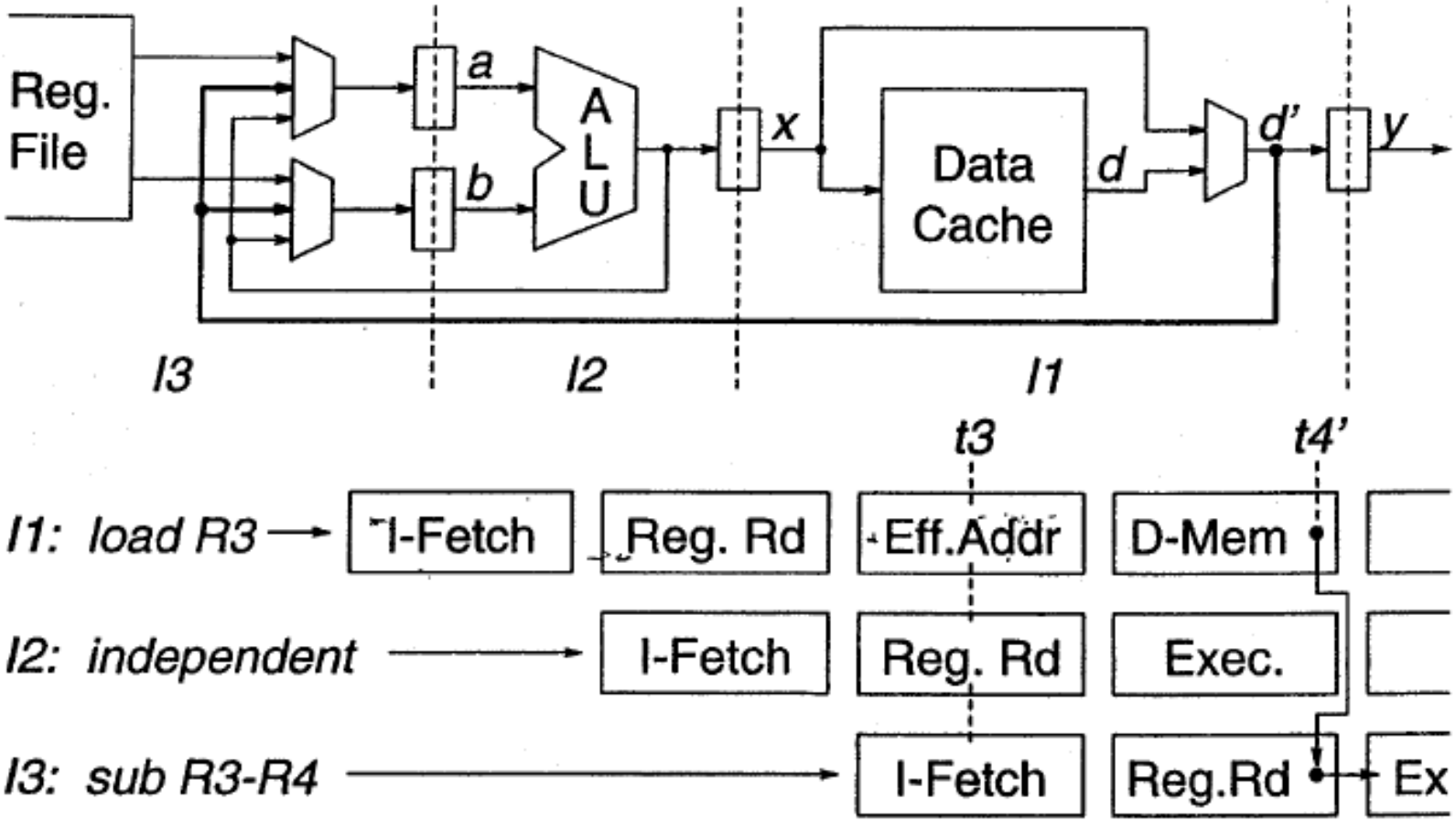

Figure 20.4 Bypassing from the fourth to the second stage

CISC/RISC Comparisons

- The execution of a typical CISC execution: *a memory-to-memory three-operand add*
  - Fig. 20.5
- Conclusions:
  - RISC is more flexible than CISC in finding and exploiting parallelism
    CISC determines and defines the exploitable instruction-level parallelism at *machine-design time*, while RISC does so at *compile time*

Comparing RISC and CISC

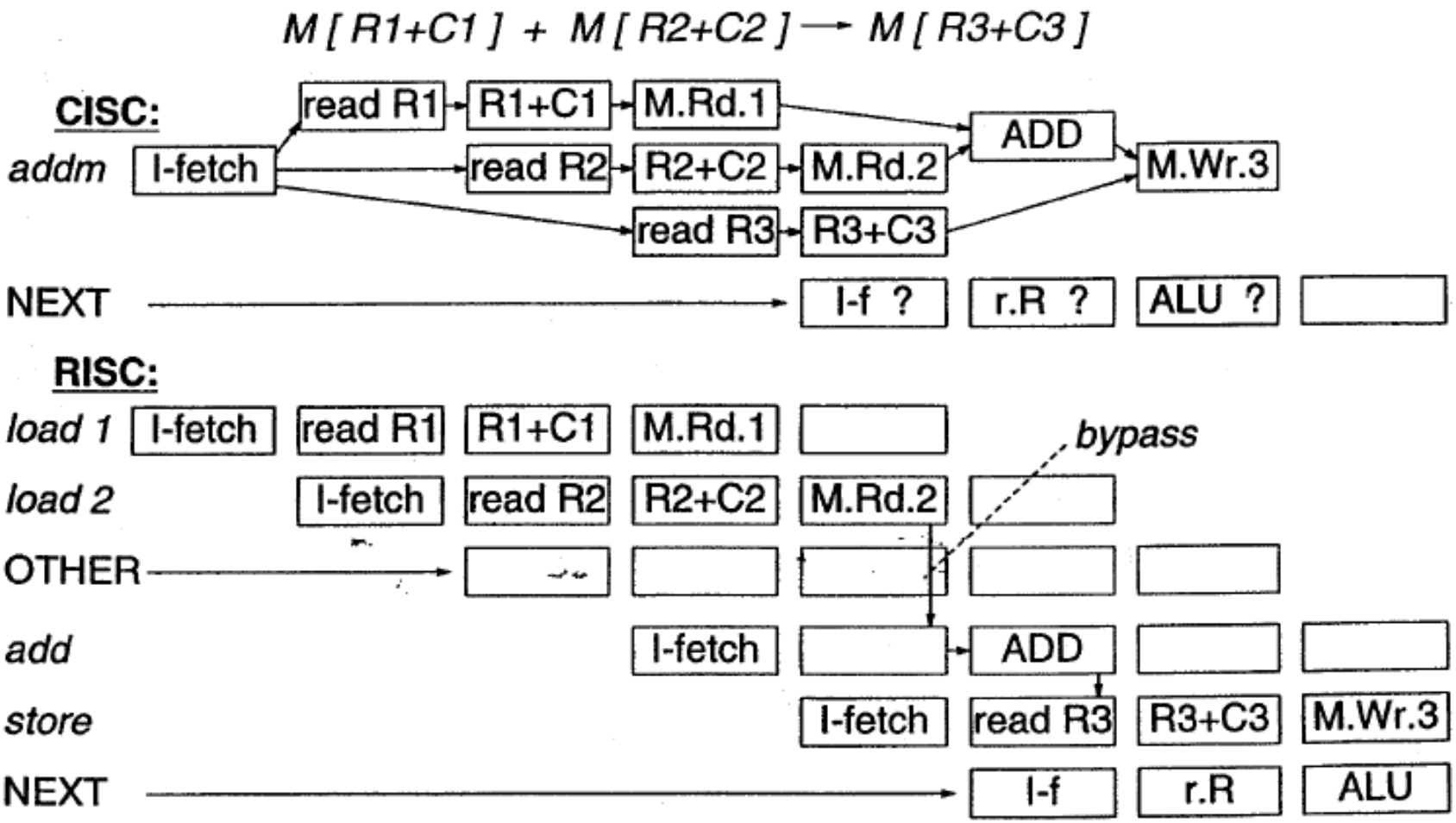

Figure 20.5 Memory-to-memory addition in CISC and RISC

Instruction Scheduling

- Rearrange instructions to *fill load delay slots*
  - Instructions need to be *independent*

- *Static*: At compile time
- *Dynamic*: At run time.
  "out-of-order execution"
  - needs considerable hardware
  - often costs one extra pipeline stage
  - useful only when it is not known at compile time whether some instructions are mutually independent

Multiple Functional Units

- Additional register file ports
- Additional ALUs
- More complicated instruction decoding and control logic

- CISC:
  - perform multiple operations within complex instructions (defined at design time)
- RISC:
  - fetch and execute multiple instructions in each clock cycle of the pipeline
    - defined at *compile time*: **VLIW** (Very Long Instruction Word)
    - defined at *run time*: **Superscalar**

Delayed Branches

- While the branch is being executed, the next instruction is already in the pipeline. What to do if the branch is successful?
  - Annul?
  - Always execute it? (Utilize the *delay slot*)
    - i.e., the instruction is always executed, as if it were *before* the branch, yet the branch instruction cannot depend on it, since this instruction is actually executed *after* the branch
- Merely another case of instruction scheduling, to be handled by the optimizing compiler
- Delay slots can be filled by instructions which are
  - moved there *from before* the branch
  - moved there *from one of the two branch targets* (consequence?)
  - *noop* instructions

RISC Summary

- RISC: The raw data path operations are exposed to and controlled by the object code generated by the compiler and the optimizer.
- RISC is the result of thorough understanding of how hardware and software best co-operate
- 1980s -- "*plain*" RISC processors
- early 1990s -- *superscalar* RISC processors
- current development -- *latency tolerant* superscalar RISC processors (switch context while waiting)