WHAT ARE THE REGISTERS FOR?

B. Goossens

DALI
Université de Perpignan Via Domitia
webdali.univ-perp.fr

Russie, août 2006

UP VD

Outline

- **ILP in the source code.**
- **Object code and IPC.**
- **Renaming and IPC.**
- **Load/store and computation.**
- **Why registers?**
- **Where the register file?**

Source code ILP

```
void saxpy( float x[], float y[], float a, uint n) {
 uint i;
 for (i=0; i<n; i++) y[i] += x[i]*a;
}
```

n independent computations.
2 reads, then 1 product, then 1 addition
then 1 write, repeat n times.

The number of Instructions run Per Cycle is limited by the number of functional units.

Object code in a RISC ISA

```
saxpy: // x in R1, y in R2, a in F1, n in R3
        CLR     R8          // i=0
        CMP     R9, R8, R3  // i==n
        BEQ     R9, end     // if (i==n) return
e:      FLD     F2, R8(R1)  // F2=x[i]
        FMUL    F3, F2, F1  // F3=a*x[i]
        FLD     F4, R8(R2)  // F4=y[i]
        FADD    F5, F4, F3  // F5=y[i]+a*x[i]
        FST     F5, R8(R2)  // y[i]+=a*x[i]
        ADD     R8, R8, 1   // i++
        CMP     R9, R8, R3  // i!=n
        BNE     R9, e       // if (i!=n) goto e
end:    RET
```

RISC ISA main features (load/store arch)

Memory data are loaded into the registers (LOAD instruction).

Computation sources and destination are registers.

Persistent results are stored from registers to memory (STORE instruction).

What are the differences between source code and assembly code?

Iteration (i) depends on iteration (i-1) though the computation of i and through the registers.

The dependence on i sequentializes the iterations starts according to the adder delay.

The dependence on the registers names sequentializes the operations according to the operators delays.

The dependence on i sequentializes the iterations starts according to the adder delay.

x[i] and y[i] can be loaded only when i has been computed, i.e. when the incrementation of R8 is produced by the adder.

If the adder has a 1 cycle latency, at most one iteration can be started per cycle. The IPC is limited by the number of instructions per iteration (unroll the loop to increase the IPC).

The dependence on the registers names sequentializes the operations according to the operators delays.

Loading x[i] into F2 may start only when F2 is free
i.e. when FMUL in iteration i-1 has read F2
i.e. when x[i-1] has been loaded.

x[i] loading may not start before x[i-1] loading is ended.

IPC <= 8/11 (0,73)

```
(1) e:   FLD    F2, R8(R1)
(2)      FMUL   F3, F2, F1
(3)      FLD    F4, R8(R2)
(4)      FADD   F5, F4, F3
(5)      FST    F5, R8(R2)
(6)      ADD    R8, R8, 1
(7)      CMP    R9, R8, R3
(8)      BNE    R9, e
```

Latencies	
FLD:	2
FST:	2
FMUL:	4
FADD:	3
ADD:	1
CMP:	1
BNE:	1

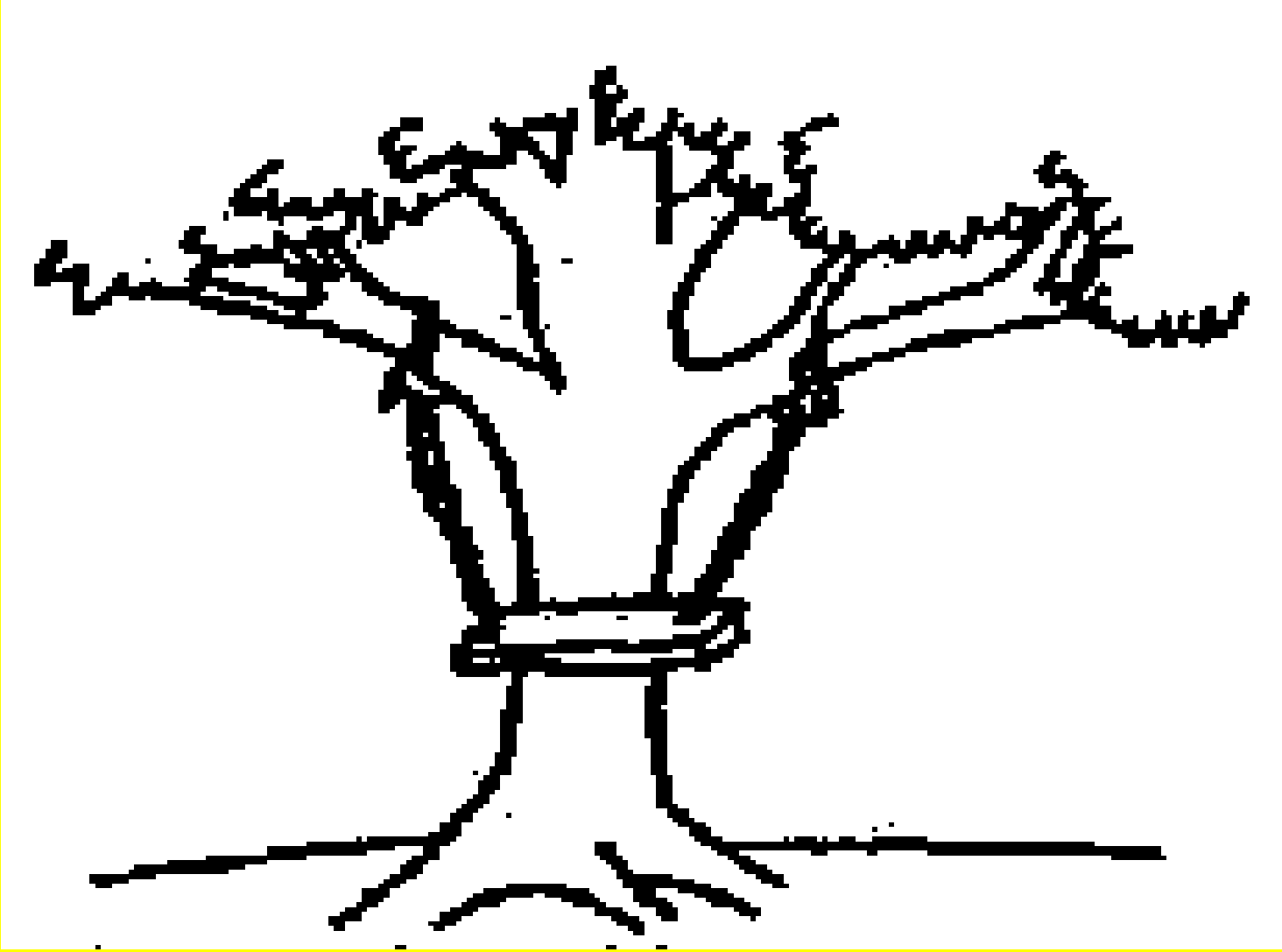
13, -, 2, -, -, -, 4, -, -, 5, 6, 7, 8

13, -, 2, -, -, -, 4, -, -, 5

UP VD

Registers create artificial dependencies

Mapping the program variables onto the set of processor registers introduces artificial dependencies into the code.
(the mapping is surjective as there are more variables than registers)



Russie, août 2006

Renaming

```
e:  FLD    F2, R8(R1)    // FLD    FF2-0, RR8-0(RR1-0)
    FMUL   F3, F2, F1    // FMUL   FF3-0, FF2-0, FF1-0
    FLD    F4, R8(R2)    // FLD    FF4-0, RR8-0(RR2-0)
    FADD   F5, F4, F3    // FADD   FF5-0, FF4-0, FF3-0
    FST    F5, R8(R2)    // FST    FF5-0, RR8-0(RR2-0)
    ADD    R8, R8, 1     // ADD    RR8-1, RR8-0, 1
    CMP    R9, R8, R3    // CMP    RR9-1, RR8-1, RR3-0
    BNE    R9, e         // BNE    RR9-1, e
e:  FLD    F2, R8(R1)    // FLD    FF2-1, RR8-0(RR1-1)
    FMUL   F3, F2, F1    // FMUL   FF3-1, FF2-1, FF1-0
    FLD    F4, R8(R2)    // FLD    FF4-1, RR8-0(RR2-1)
    FADD   F5, F4, F3    // FADD   FF5-1, FF4-1, FF3-1
    FST    F5, R8(R2)    // FST    FF5-1, RR8-0(RR2-1)
    ADD    R8, R8, 1     // ADD    RR8-2, RR8-1, 1
    CMP    R9, R8, R3    // CMP    RR9-2, RR8-2, RR3-0
    BNE    R9, e         // BNE    RR9-2, e
```

IPC <= 8

```
(1) e:   FLD    F2, R8(R1)
(2)      FMUL   F3, F2, F1
(3)      FLD    F4, R8(R2)
(4)      FADD   F5, F4, F3
(5)      FST    F5, R8(R2)
(6)      ADD    R8, R8, 1
(7)      CMP    R9, R8, R3
(8)      BNE    R9, e
```

Latencies	
FLD:	2
FST:	2
FMUL:	4
FADD:	3
ADD:	1
CMP:	1
BNE:	1

```
13, -,  2,  -,  -,  -,  4,  -,  -,  5
6,  7,  8
    13, -,  2,  -,  -,  -,  4,  -,  -,  5
    6,  7,  8
```

Renaming is handled by the hardware

The hardware allocates and frees the renaming registers.

The hardware allocates a register from a free list. A register is freed when the instruction ends.

The number of renaming registers constrains the ILP.

Renaming increases the register file area

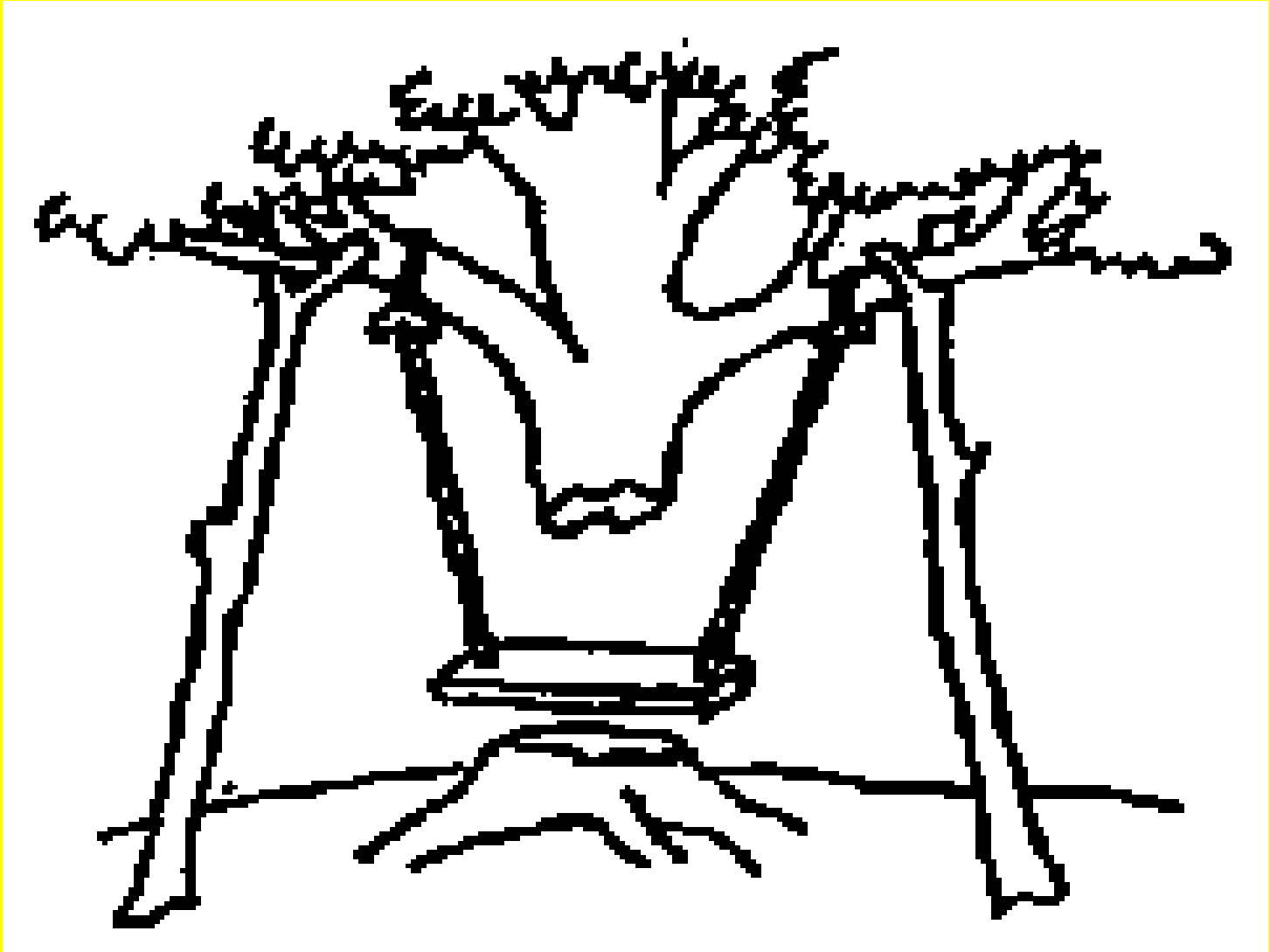
The renaming registers set is an expansion of the architectural registers set.

The bigger the renaming register file, the more ILP the hardware can capture but the higher the register file access time.

Renaming

Renaming is an expansion of the compiler registers set into the processor registers set.

The compiler maps the variables onto the architectural registers and the hardware relaxes the centralization induced by expanding the architectural registers onto microarchitectural registers.



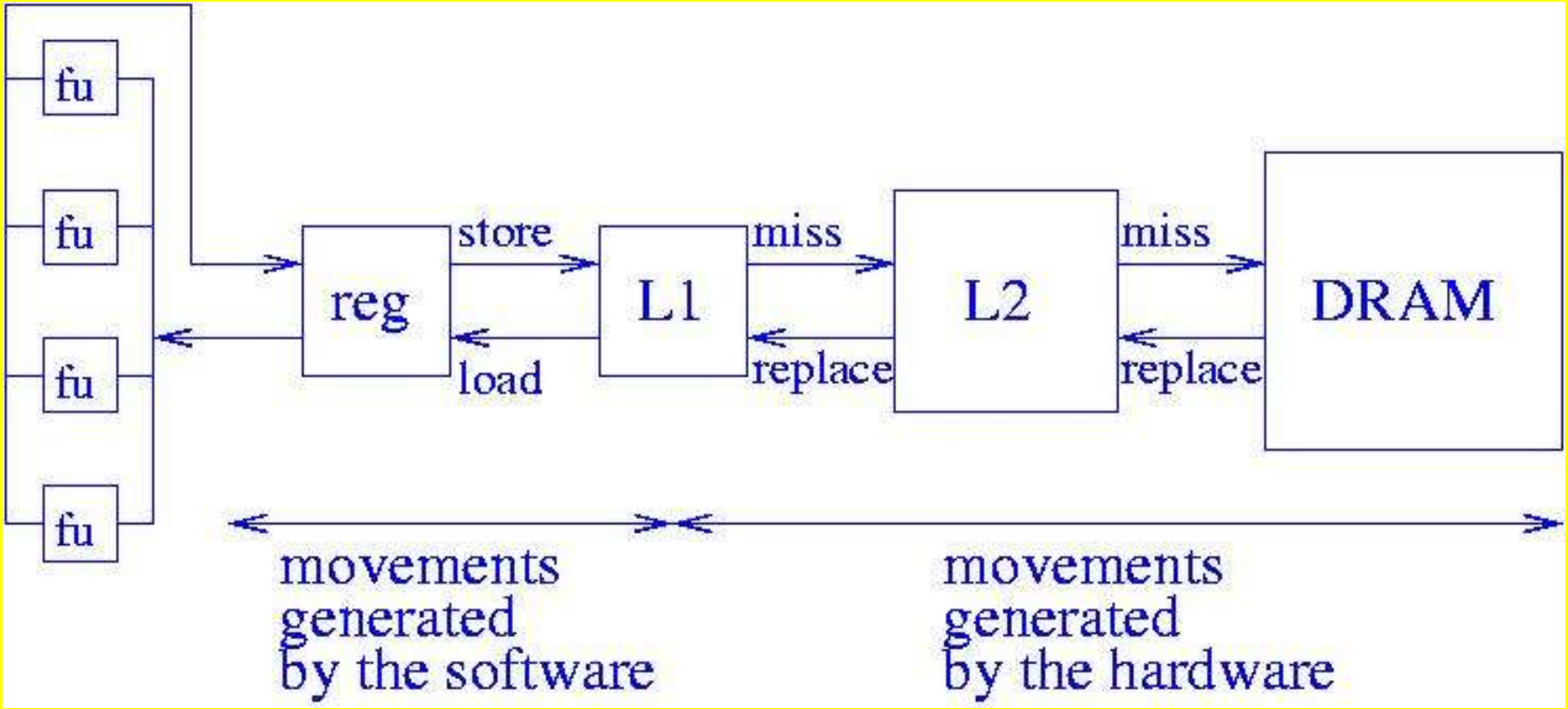
Russie, août 2006

RISC architecture (load/store)

```
a = b + c;      LD    R1, b
                LD    R2, c
                ADD   R3, R1, R2
                ST    R3, a
```

The memory hierarchy hardware takes care of the data migration, launched by specific loads and stores instructions inserted by the compiler into the object code.

The data path

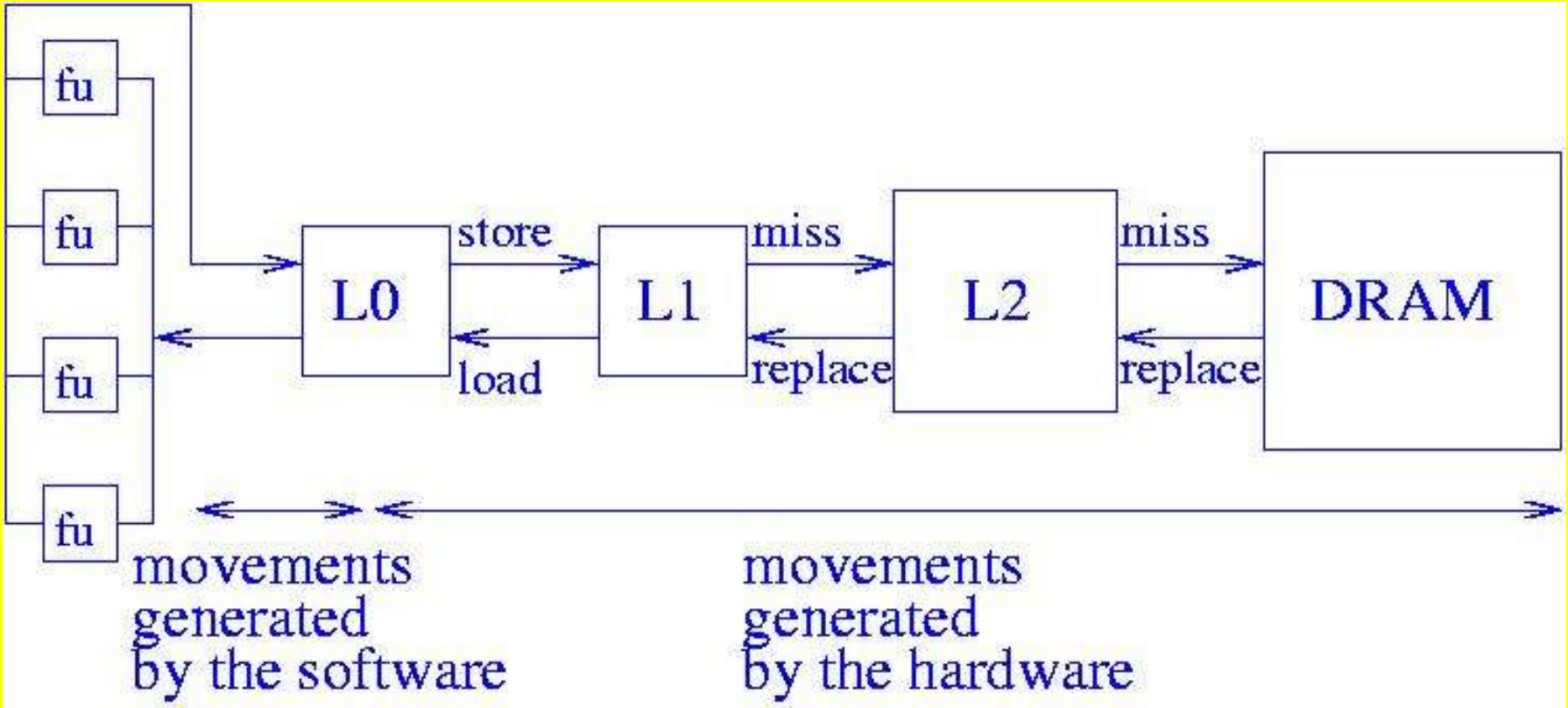


Memory/memory Architecture

a = b + c;

The most direct path is to read from b and c and to write into a.
The memory hierarchy hardware moves the sources from the closest up-to-date copy to the adder and then moves the result back to the first memory level.

An L0 cache in place of the register file



Why registers?

Answer n°1: a register file is a faster memory than a cache.

Partially wrong answer: due to the multiported structure of the register memory cell, its area is larger than the cache memory cell.
Partially right answer: a register file allows parallel accesses to any set of registers. A multibanked cache allows parallel accesses to the banks set.

Why registers?

The register memory cell area grows according to the square of its access ports.

Within the same area, one can design a 16-ports n bits register file or a 64 banks n bits 2-ports cache.

Within the same area, a 16-ports register file contains n bits and a 2-ports cache contains 64*n bits.

Why registers?

Answer n°2: they simplifie the object code.

Most appropriate answer.

Registers are a renaming technique and not a storing technique. They exhibit (statically) the data dependencies. Unfortunately, they also introduce false dependencies.

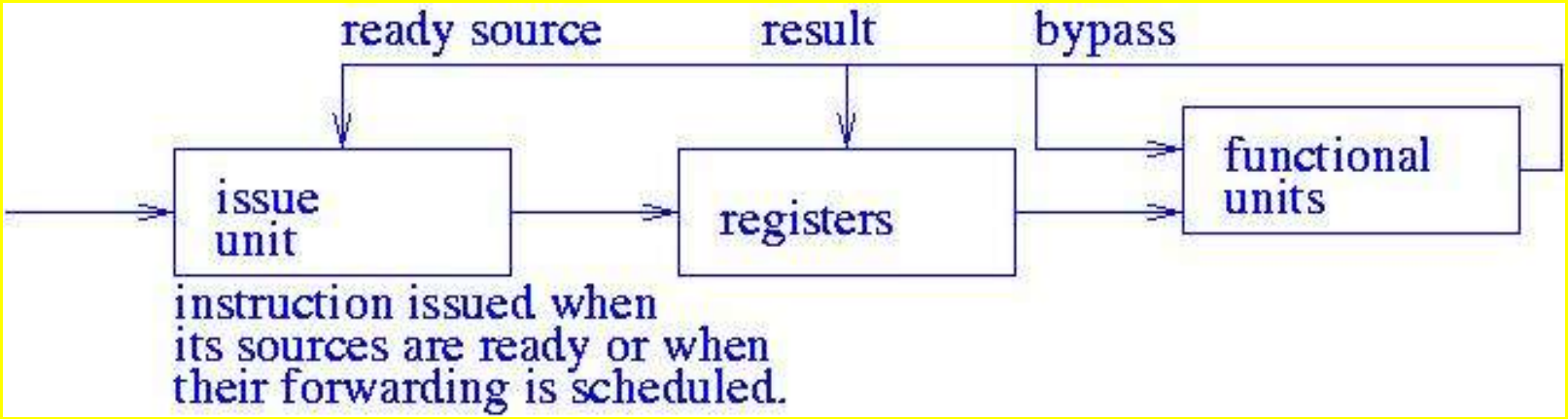
Why registers?

Answer n°3: they exhibit (statically) the intermediate (volatile) results.

A register holds an intermediate result if it is reused as the destination of a later instruction before the result is written.

Intermediate results should only be forwarded to the consuming successors and not stored. This saves both read and write ports.

Actual data path

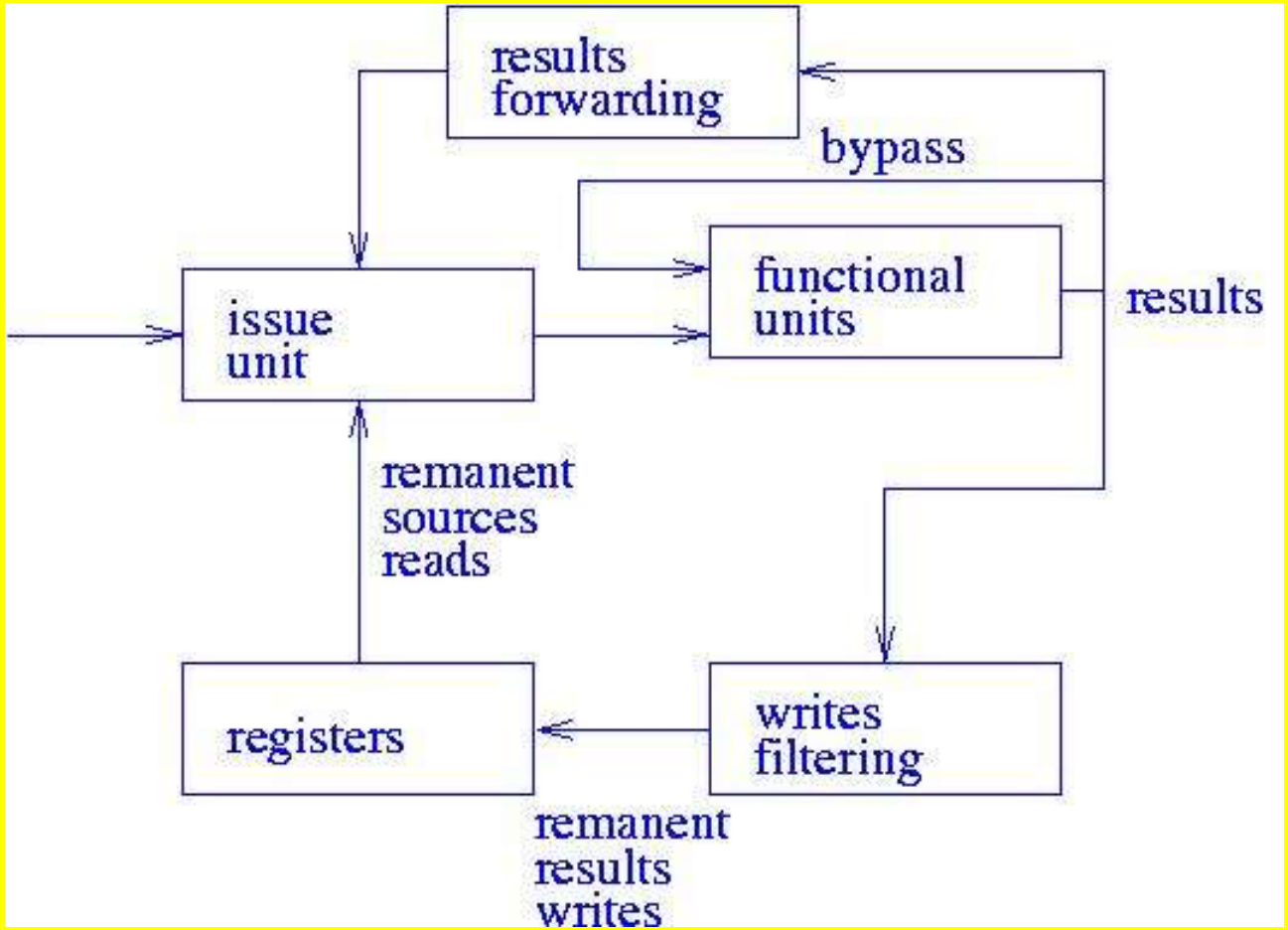


All the results are written to the file.

All the sources are read from the file (even if bypassed).

Not suited to high superscalar degree.

Proposed new data path



From previous talk:
1 write port,
7 read ports,
16 issues per cycle.

Volatile results are forwarded or bypassed.

Remanent results are stored and later reloaded.

Main structures: write filter, forwarder, scheduler.

UP VD

Register name semantic

The compiler renames memory variables with register names. It does so through a load instruction.

A load inserts a memory point into the data path.

A remanent result is removed from the data path into the memory space. The register file is mapped on the process virtual memory space.

A store moves a datum from the register memory space to the program variables memory space.

Register name semantic

```
LD      R8, 0(R1)     //inject b in the path, named R8
LD      R9, 0(R2)     //inject c in the path, named R9
ADD     R10, R8, R9
ST      R10, 0(R3)    //move mem R10 to mem a
```

This semantic does not make any difference between registers and memory. Registers saves and restores at context switch are automatic. Multiple register files for multiple threads can be handled.

Renaming

The hardware does not rename memory mapped registers.

Instead, it assigns a volatile name to every in-flight instruction (in a circular FIFO fashion).

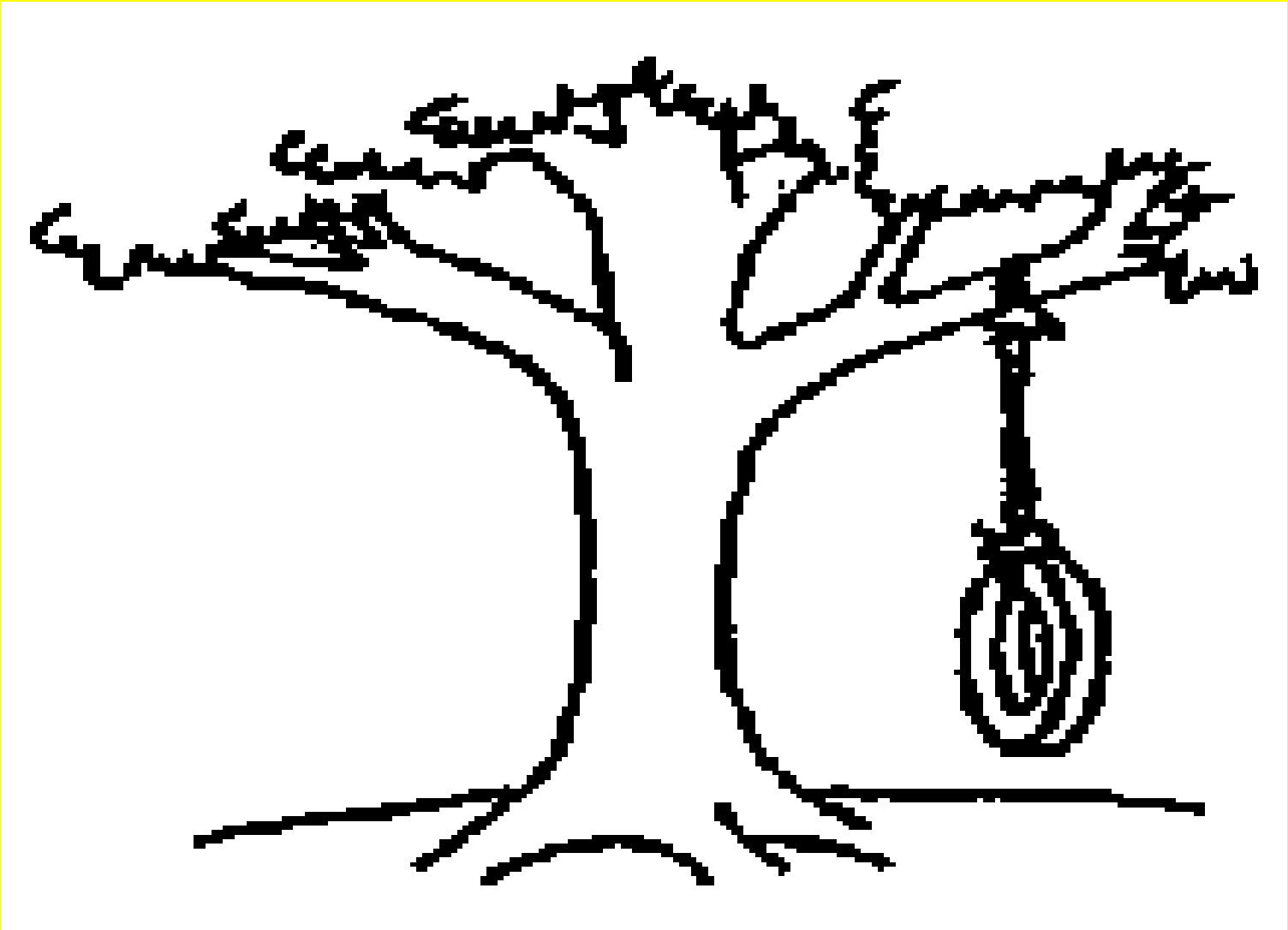
RAT[i] keeps the name of the most recent instruction with destination register i.

Renaming

```
LD      R8, 0(R1)     //instruction i1 writes to R8
LD      R9, 0(R2)     //instruction i2 writes to R9
ADD     R10, R8, R9 //R8 from i1 and R9 from i2
ST      R10, 0(R3)    //R10 from i3
```

The scheduler knows where a source is (not yet computed, forwarded or written).
If it is not yet computed, the scheduler knows where and when it should be output.

Conclusion



Russie, août 2006