Project Name: OpenRisc 2000

Project's Mission
We want to build processor core, that allows us greater clock speed and at the same time more Dhrystone MIPS / MHz. NOTE: Since I'm a software engineer, I may use inapropriate or wrong terms, so please have patience ;)

History
It looks that computer hardware (silicon) will reach its limits in few years, and current arhitectures could not be easily improved. Software enginering has moved and is still moving to higher languages. Languages highly dependend on arhitecture are less and less used. All that leads us into search for new arhitecture. Many solutions we tested and since this arhitecture is still under development, we will present the current one.

Description
Analises showed, that instructions are normally not dependent on all registers in register file. Since the many hazards which come with simultaneus access to several registers, we proposed an idea, based on systolical array computers. The idea is to provide as less register visibility (from *functional unit* point of view) as possible.

Functional unit (later FU) can do any logical or arithmetic integer operation like AND, OR, SHL, MOV, MUL (this can cause some problems if MUL is too slow, but at first design we can skip it, and simulate MUL). Note, that every FU does own simple instrucion decoding.

FUs are connected locally with optimized connection topology (so buses can be short). Registers are distributed, so that each FU includes only one register, into which it can write. Also the data from his *neighbours* could be read. *Neighbours of x* are all FUs, that is *x* connected to.

But there is one problem - we normally need more registers than we have FUs. We solved this problem using *local registers* as described later. There are also some great benefits of such local processing - no pipelining and that also means all those pipelineing hazards are history (note, that we may need pipelineing just for instruction loading, if we have data and instruction coming on the same bus).

What matrix size should we choose? Following instruction appearance frequencies, aproximately 10%of memory store, 20%of memory load and 70%of others, we conclude that, 3 to 4 FUs on the same bus could efficiently access memory (in average and by appropriate compiler optimization). For example 3×3 matrix would look like:

Besides local instruction decoding (done by each FU), we have also global decoding unit, which decodes general instructions, and passes data into FUs. General instructions include jumps and other control type and some special instructions.

Since there is a limit in parallelity of current programs, we can estimate, that larger vertical matrix sizes would not yield more speed.

Local Registers
If we add to each FU_i some local registers L_{ij} which can be access only by

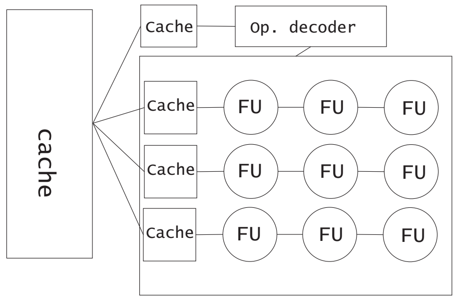

Figure 1: Block diagram

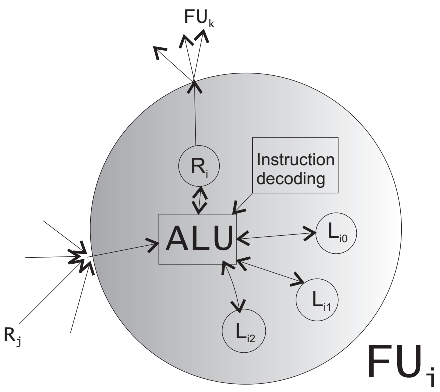

Figure 2: Functional unit

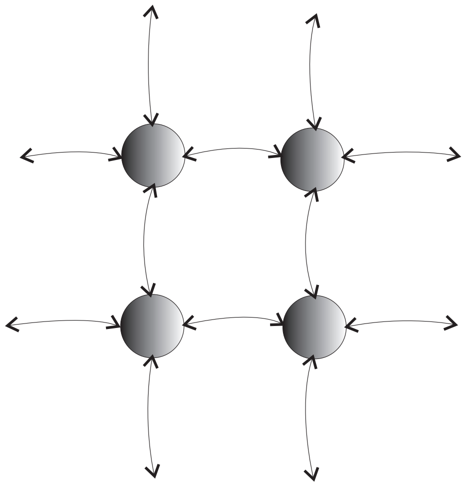

Figure 3: Basic Topology

FU_i, we do not need to add more connections between FUs. Compiler can be also easily designed to use full power of these registers, as described in compiler section. Addressing of such registers for FU_i with main register R_i would look like:

$$\{operator\}\{destination : L_{ij} \vee R_i\}\{source : L_{ik} \vee R_i\}\{source : L_{ik} \vee R_n\}$$

where R_n is register of a neighbour (FU_n is neighbour of FU_i).

Topology

It is very important, that we chose good topology (to allow FUs maximum possible communication with smallest possible connections). Also there is one request - make all FU equivalent (such, that we do not know in which FU we are just by knowing connection types), which allows us better code optimization algorithms. Following picture shows two basic topologies where all FUs are equivalent. First one includes only red connections, and heksagonal both red and blue.

How Does it Work ?

Generally all FUs execute instruction in each and one cycle. If such instruction loading would be too hard to do, we can make compromise: we can load whole matrix rows or columns at a time:

$$\{row\}\{\{instr_0\}\{instr_1\}\{instr_2\}\}$$

and at same time we save some space (it is very important that we reduce command flow to minimum, since now even greater memory/processor speed ratio). Note, that there are no arithmetic flags, however there may be something like control word register and special flags which allows efficient loop termination. Why is it smart to use many addressing types? Memory is slow - each extra access is redundant and significally decreases performance, and besides, it's only a minor complication, since we need it just for load/store instructions.

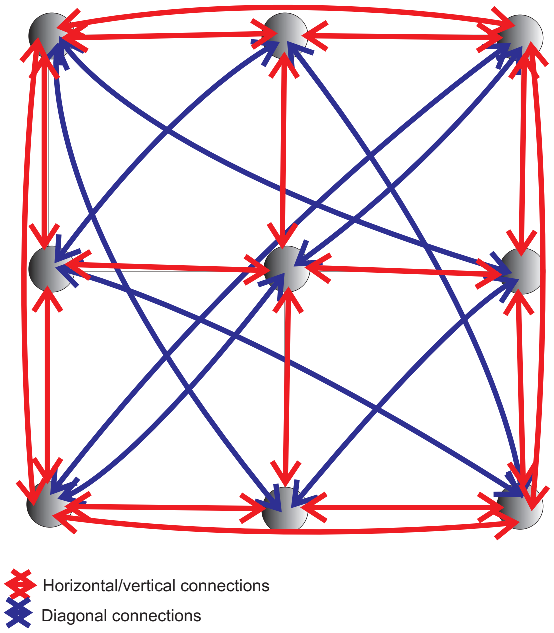

Figure 4: Metal layers

Compiler

Because we designed software on hardware (almost), we can expect more complicated compiler than current ones. We are fortunate (well, we designed it so), that this arhitecture is pretty similar to ordinary RISC arhitectures, so in first stages we can program with a bit modified GNU GCC compiler. Tests showed, that using GCC with properly defined *register classes*, we can make pretty good output code. However, for optimum results we can use algorithms developed for this application. We also think that, after some time, optimized program in assembly code could be written just as fast as on ordinary pipelineing computer.

Local Registers

When given program data flow graph, we can convert register usage: we can store register R_i into local one L_{ij} instead in R_i and gain one free register, and by next access of (previous) R_i, we just use L_{ij}, as shown on Figure .

Cache Line Loading

There is a possibility to include some special instructions that loads whole cache line into defined registers. Currently we are not working on this idea, since it will probably complicate compiler. Maybe after some time such improvement would be nice, bacause it would singificantly increase memory performance, allowing us more FUs on the same bus.

Code Optimization Algorithm

Adapting current compiler optimizing schemes, makes things pretty complicated. So we propose different point of view. Optimize program execution as we would have enough registers. Optimal height enumeration as shown on figure takes linear time $\mathcal{O}(n)$. Then we shrink the graph. Width is defined by num-

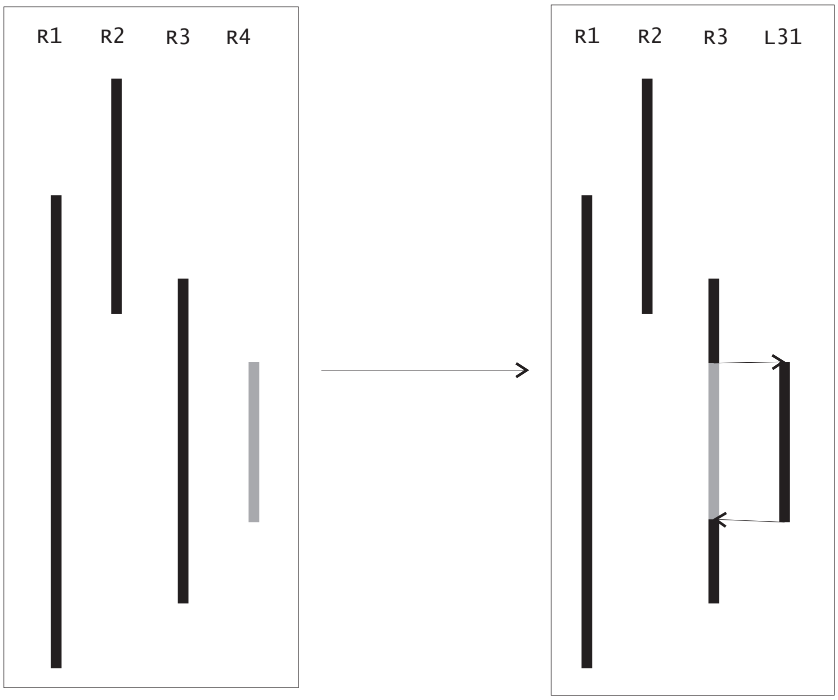

Figure 5: Local Register Usage

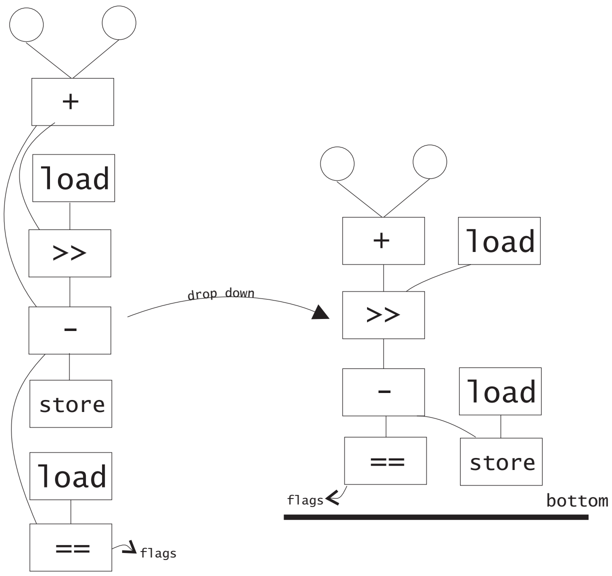

Figure 6: Finding dependencies

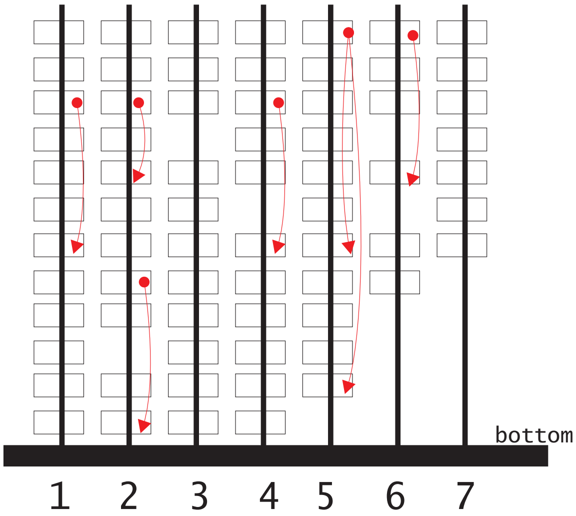

Figure 7: Allocating resources

ber of FUs. This problem is $\mathcal{NP}$-complete, so we have to use an aproximation algorithm. Having done that, all we have to do is to allocate registers. Each FU allocates only one register, so part of register allocation is already done. From now on we propose following algorithm: we go from bottom up and permutate current instructions (that have $height = i$), so every instruction below us can access its data. If such rearangment is not posible, we use local registers, or if even that fails, we spill register and enumerate height of upper graph again. We can expect, that this (greedy) aproximation algorithm ($\mathcal{O}(nm)$, where m is number of FUs) will yield good results, since many natural gaps in graph allows several free data acceses.

Efficent loop unrolling

Simulations have shown, that basic blocks (sequence of code, with one input and one output point) in normal desktop computer can rearange instructions to archive in average 2.09 IPC. With some OpenRisc 2000special improvements, we can archive up to 3.19 IPC. What about other FU? We can use SMT design as described later or we can significantly boost performance using efficent loop unrolling. Suppose we have a statement in high level language:

`for(x;y;z1) z2;`

Compiler would translate this to something like:

```
loop:  x
cond:  y
       jump cond,end
body:  z1
       z2
       jump cond
end:
```

If loop is natural and at loop start we know how many times will it repeat, efficient loop unrolling would look something like:

```
opt_ loop:  preloop execute (k-1 instructions)
opt_ cond:  jump count >= 0,opt_ end
opt_ body:  multiple pass (k passes) loop unrolled instructions
            jump opt_ cond
opt_ end:   postloop execute (k-1 instructions)
```

Where k is height of z1 $\cup$ z2 graph.

Simultaneous Multi Threading (SMT)

If efficiency of FU, will be too low, even after loop unrolling, and we still want to achieve greater IPC ratio, we can apply SMT idea to OpenRisc 2000. Suppose that we have m threads and $k \times l$-matrix. In each cycle there exists a priority numbering $p_i = (i + c)_{modm}$, where c is number of current cycle (lower the p_i, higher priority). Then FU matrix is filled by descending thread priority - and of course only non-ocupied FUs in matrix can be filled. Note also, that there is no "FU renaming" for example, if t5 needs FU 4, it waits for that FU to be free, it cannot use any other (eg. FU 7). If compiler randomly distributes FU usage, we can get pretty good average (when $m \to \infty$: efficiency $\to 1$). Of course we must also have in mind that Fu cannot be allocated, if it accesses registers of any other thread FUs, unless we add some more harware logic in FUs. Each thread instruction matrix is executed and can execute next one, when all instructions from the instruction matrix have been executed.

This resource sharing scheme is actually improved Round-Robin, and guaranties us, that each thread executes at least one instruction matrix per m clocks.

Sections Coming:

Instruction ordering in global optimization
Flags
Branch Prediction
Hardware Implementation
Exceptions
Memory Organization

Last updated: December 17, 1999