THE EFFICIENCY OF THE REGISTER FILE BASED ARCHITECTURES IN THE OOP LANGUAGES ERA

Marius Stoian & Gheorghe Stefan

Department of Electronics, Politehnica University of Bucharest,
E-mail: gstefan@arh.pub.ro

Abstract: Stack oriented architectures are compared with register file oriented architectures in order to decide what is the best for building cellular programmable machines. Area & power vs. computing performance are investigated considering two kind of ``users": (i) a compiler, which is another machine, and (ii) a human mind, writing hand coded programs.

Keywords: Computer architecture, Object Oriented Languages

1. INTRODUCTION

Cellular computing replicate the same processing element (PE) in different configurations in order o increase the number of GOPS using as small as possible area and power. Any optimization is welcome if the resulting machine is designed for the consumer market. Thus, early architectural decisions are very important in the process of optimizing the area & power vs. performance ratio. One of these early decisions to be made is between the stack oriented or register file oriented architecture for the PEs used to implement a multi- or many-processor array.

The file oriented architecture looks to be more flexible, while the stack oriented one is too rigid in optimizing the number of clock cycles needed to perform a certain computation. But, when we say ``more flexible" or ``too rigid" we are referring to a human user who is involved in optimizing ``manually" the code. In real life a compiler is the one who is involved in generating the code to be executed. When the compiler becomes the user, too much flexibility starts to be a limitation and a more rigid allocation of variables starts to be a ``virtue". We expect a compiler to be more ``comfortable" with a stack architecture than with a register file architecture. We define for our purpose two simple execution units, maximizing the storage component and taking into account the loop closed through an usual arithmetic & logic unit.

The first uses a stack to store the operand and the second uses a file register for the same purposes. The two structure are synthesized in identical conditions and the area and power is measured.

Next, the two structures are used to compute the same thing executing a code generated automatically by two different (but similar as performance) compilers. The number of clock cycles will be compared.

We will state preliminary conclusions based on this simple and short investigation. The result will be used to encourage more detailed, focused, and sophisticated investigations.

The second approach investigates a special kinds of stack in comparison with the same file register. Because no compiler is available, only hand coded

assembly programs are used to evaluate the performance.

2. THE TWO STRUCTURES

Two test structures are considered for evaluating the architectural and structural effects of the alternative fileregister/ stack. Both are designed to store the same number of variables (16), one in a file register and another in a stack. The two structures use the same arithmetic & logic unit (ALU).

2.1 Register file based execution unit

The file-register version stores the variables in a 2-output one-input 16-word file-register. For the sake of simplicity the following Verilog description uses compact coded inputs and outputs. The execution unit consists in an ALU, loop connected with the associated file-register. In order to avoid any ambiguities the most accurate description of the structure is provided as a Verilog module.

```
module file_version( input                clock,
                     output  reg [18:0]  out    ,
                     input       [50:0]  in     );
  reg    [50:0]     in_reg                ;
  reg    [2:0]      flags                 ;
  wire   [15:0]     left_op, right_op     ,
                    result, reg0          ,
                    left_out, right_out   ;
  wire   [2:0]      alu_flags             ;
  always @(posedge clock)
    begin  in_reg  <= in      ;
           out     <= {alu_flags, reg0};
    end
  file_reg file_reg(.left_out      (left_out      ),
                    .right_out     (right_out     ),
                    .register0     (reg0          ),
                    .left_addr     (in_reg[35:32] ),
                    .right_addr    (in_reg[40:37] ),
                    .dest_addr     (in_reg[45:42] ),
                    .in            (result        ),
                    .write_enable  (in_reg[46]    ),


                    .clock         (clock         ));
    mux2_16 left_mux(.out(left_op      ),
                     .in0(left_out     ),
                     .in1(in_reg[15:0] ),
                     .sel(in_reg[36]   )),
            right_mux(.out(right_op                 ),
                      .in0(right_out                ),
                      .in1(in_reg[31:16]            ),
                      .sel(in_reg[37] & in_reg[41]));
    alu alu(.left      (left_op      ),
            .right     (right_op     ),
            .alu_func  (in_reg[50:47] ),
            .alu_flags (alu_flags    ),
            .out       (result       ));
endmodule
```

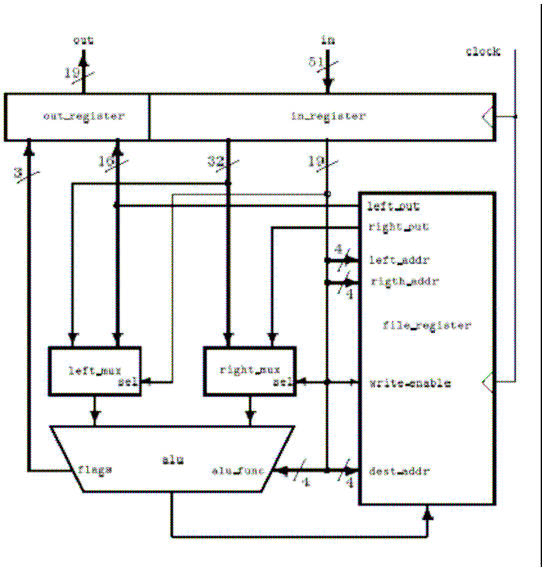

Fig. 1. File-register execution unit.

In each clock cycle two operands are fetched from the file register to the input of ALU, and the resulting output of ALU is written back into the register file.

2.2 Stack based execution unit

The stack version for the execution unit is considered in two versions. A simple one is provided to be associated with the behavior of a Java compiler, and a more complicated one will be associated with ``hand written'' code.

The operations performed by the stack are the following: nop, push, pop, write, pop_write, swap. The module incorporates the same ALU used by the register file version. The interface registers are similar (minus 10 bits for the input register, because less bits are needed for controlling this kind of unit). The associated Verilog module for the simple version follows.

```
module simple_stack_version(input                clock  ,
                            output  reg [18:0]  out    ,
                            input       [40:0]  in     );
  reg    [40:0]  in_reg             ;

  wire   [15:0]  stack0, stack1;
  wire   [15:0]  result             ;
  wire   [2:0]   alu_flags          ;
  always @(posedge clock)
    begin  in_reg  <= in                ;
           out     <= {alu_flags, stack0;
    end
  simple_stack stack(.stack0 (stack0        ),
                     .stack1 (stack1        ),
                     .in0    (result        ),
                     .in1    (stack0        ),
                     .func   (in_reg[36:34] ),
                     .clock  (clock         ));
  alu alu(.left (in_reg[32] ? in_reg[15:0] : stack1   ),
          .right(in_reg[33] ? in_reg[31:16] : stack0),
          .alu_func   (in_reg[40:37]                  ),
          .alu_flags  (alu_flags                      ),
          .out        (result                         ));
endmodule
```

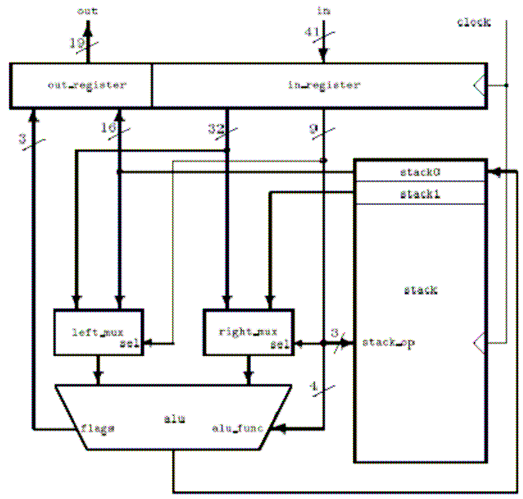

Fig. 2. Stack execution unit.

The stack module is a 16-level simple stack with access to the first two recordings: stack0, stack1. The binary operations use stack0, stack1, while the unary operations use stack0. The result is pushed or written back in top of stack (stack0). The operation swap acts on stack0, stack1.

Because the operands are stack0, stack1 the speed is expected to be improved because the fetch of the operands, from the register file version, is avoided.

This kind of execution units has *20%* less control bits than the previous.

3. EVALUATIONS

The solutions for the two execution units are compared taking into account the physical resources involved (area & power) and the resulting performances (speed & clock_cycles/task).

3.1 Area & Power

The two structures are synthesized for *130 nm*, standard process with the same time constriction of *2.5 ns* (the timing limit is imposed by the file-register version which is the ``laziest").

The file register version uses *83%* more area and consumes *55%* more power (the power is estimated considering that *12%* is the mean degree of structures switching in each clock cycle). Results a very big structural advantage for the stack version. The effect of this advantage will be partially diminished by the decreased performances in execution achieved by the stack version. It depends on how big is the increase of computing time expressed in the number of clock cycles.

Table 1 The file register vs. the stack version with the time restriction of *2.5 ns*.

Version	Area	Power	Time
File register	44602 μ^2	2.32 mW	2.51 ns
Stack	24320 μ^2	1.49 mW	2.35 ns
File/Stack	1.83	1.55	

3.2 Computing performance

A first experiment consist of the same computation performed on both structures. Let be the computation described by the following expressions (the test is provided by [3], and can be considered meaningful for the purpose of our investigation):

a <= a + b + c + d/2;
b <= a + b/2 - c - d;
c <= a - b/2 - c + d;
d <= a - b + c - d/2;

The associated C program, compiled for a file register machine, is:

```
int f(int a, int b, int c, int d);
int main(void){
f(1234,5678,9012,3456);
return 0;
}
int f(int a, int b, int c, int d){
int a1 = a + b + c + d/2;
int b1 = a + b/2 - c - d;
int c1 = a - b/2 - c + d;
int d1 = a - b + c - d/2;
return g(a1,b1,c1,d1);
}
int g(int a, int b, int c, int d){
return a + b + c + d/2;
}
```

The resulting code is executed in *28* clock cycles. Some optimization are detected when the generated code is inspected.

The Java program for the same computation, compiled for a stack architecture, is:

```
public class Test{
public static void main(String[] _args){
f(1234,5678,9012,3456);}
public static
int f(int a, int b, int c, int d){
int a1 = a + b + c + d/2;
int b1 = a + b/2 - c - d;
int c1 = a - b/2 - c + d;
int d1 = a - b + c - d/2;
return g(a1,b1,c1,d1);}
public static
int g(int a, int b, int c, int d){
return a + b + c + d/2;}}
```

The resulting code is executed in *39* clock cycles. Examining the code generated by the compiler no optimization is detected. The number of clock cycles associated to the stack architecture is *1.39* times bigger.

Theoretically, on the same area the stack version performs *31.65%* more computation.

3.3 Improved stack experiment

A second experiment will start from an improved stack structure called pseudo-stack. Hand coded programs, for the same computation, are provided for both the file register version and the pseudo-stack version.

The pseudo-stack structure is a modified stack which provides access to the first *8* levels, instead of only the first *2*, as for the simple stack previously used. The connections of the modified stack are the following:

pseudo_stack(stack0, stack1, stack2, stack3,
stack4, stack5, stack6, stack7,
in0, in1,
func,
clock);

The right operand will be selected by a *3*-bit code between the a value from the input register and one of the outputs stack1, stack2, stack3, stack4, stack5, stack6, stack7.

Thus, the pseudo-stack gains some file register features (Table 2).

For this experiment we will provide hand written code for both versions.

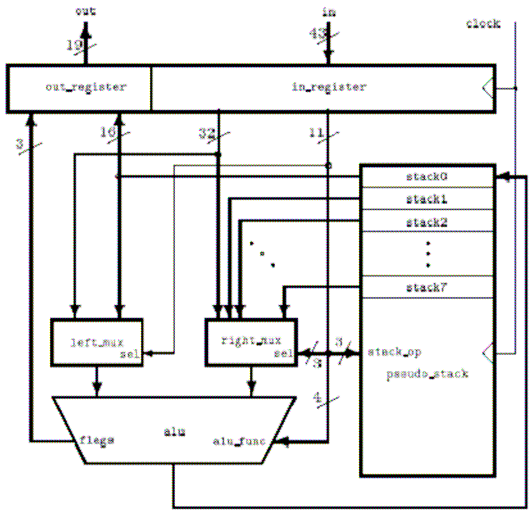

Fig. 3. Pseudo-stack execution unit.

Table 2 Adding the improved stack

Version	Area	Power	Time
File register	44602 μ^2	2.32 mW	2.51 ns
Pseudo-stack	27747 μ^2	1.54 mW	2.44 ns
Stack	24320 μ^2	1.49 mW	2.35 ns
File/Ps-stack	1.60	1.50	
Ps-stack/ File	1.14	1.03	

Results for the file version a *10* clock cycle execution and for the pseudo-stack version a 12 clock cycle execution. The file register version is *20%* faster but *60%* bigger.

Theoretically, on the same area the pseudo-stack version performs *33.33%* more computation.

4. CONCLUSIONS

The investigation presented in this paper is done in the context of the strategic switch of the consumer market toward parallel computation. Both, multi- and many processor approach (Shekar Y. Borkar, et. all , 2005) benefit from the result of this research. The PEs having small & simple execution units, considered mainly in the seminal paper about the "*13 dwarfs*" (Krste Asanovic, et. All, 2006), can be optimized using the conclusions of our research.

1. The main result of our investigation is: stack oriented architecture has a big chance to improve with at least *30%* the use of the area in cellular (many-processor) computation.
2. The number of wires (data & control) broadcasted into an array of stack processors is significantly reduced (from *35* to *25* or *27*) allowing important savings in area and power. For a file-register version *30 - 40%* more wires are used.
3. The power is more efficiently used by stack units, but not as much as the area, because inside the stack the bits ``are moving" at each push or pop.
4. The pseudo-stack reduces the cycle performance gap between the file register oriented architecture and the stack oriented architecture. The "price" for this is very small: *14%* in area, and almost nothing in power. The power is almost the same because data moving in stack is the same in both versions.
5. There are also a lot of effects on efficiency in the generation of the code for a stack machine. But these effects are beyond thepurpose of this investigation.

REFERENCES

Krste Asanovic, et. All 2006.: *The Landscap of Parallel Computing Research: A View from Berkeley, Technical Report No. UCB/EECS-2006-183*, December 18,

Shekar Y. Borkar, et. all. 2005: *Platform 2015: Intel Processor and Platform Evolution for the Next Decade*, Intel Corporation,.

Bogdan Mitu: Private Communication.