The Tiny CPU Generator (TCG)

Yosi Ben-Asher
Computer Science Dept.
Haifa University, Haifa, Israel
yosi@cs.haifa.ac.il

1 The instructions set

The machine:

The machine is specified by several pipelines where each pipe is activated by a different instruction forming a VLIW type of instructions.

- Each instruction is n bit long where n is determine by the coding of k pipeline stages. For each pipeline, there is a separate memory bank that hold the instructions destined for this pipe. Thus, the instruction of a VLIW $\alpha \mid\mid \beta \mid\mid \gamma$ are stored in three separate memories and are fetched in parallel in one clock cycle. There is a single data memory bank which is a multi-port memory bank. [1] Maximal number of pipelines (pipes) and parallel instructions is given by some constant.
- n can be freely selected though it should not increase 256 bits. Note that the more bits we use to code instruction the Decode operation is simplified and the clock cycle improves.
- The computational instructions are described by a directed graph of operations and arguments (as described in figure 1. The problem of selecting suitable coding of such directed graphs is left open.
- We use 32-bits general registers that are used for computations and for passing parameters to functions. Each pipeline can access all 16 registers.
- A special register PC points to the next instruction that should be fetched.
- Each register is backed-up by a small memory bank that acts as a stack for this register. Thus, when a function is called each register is automatically saved on its stack and automatically restored when the function returns. We assume that the recursion depth is limited to some constant.
- There is no SP and stack area to pass parameters and hold local variables. Parameters are passed via registers, though it is possible to use the main memory to pass arrays, however this requires to use explicit load/store operations to do that.

[1] You should also synthesize these memory banks as a separate memory banks.

- Each instruction must update one register (even if its a store operation) that is written back to the register file at the last stage of the pipe.
- A mechanism to stall instructions in the pipe if a value they need on a current stage will be ready at a later stage is usually needed. However, we assume that the compiler computes the amount of stalls and has inserted NOPs between instructions when needed.
- There is no need for interrupt handlers since we assume that the programs is already loaded to the pipes' instruction memory banks. A special reset signal is set to one on the first clock cycle so that initialization can occur.

There are four types of instructions as depicted in figure 1:

NOP- is an all zero instruction.

STOP- a sequence of 10 NOPs halts the computation.

Compute instructions- We assume that we have a pipeline with k stages where each stage can be either:

$+, -, *, /, <, \ldots$**-** an arithmetic operation followed by possibly shift-left, shift right, negation, zero operations $Sl/Sr/N, Z$.

E **-** Any possible arithmetic operation.

C **-** Any comparison operation.

B **-** Any boolean operation.

M **-** Load or store operation. If there are more than one M stages than we assume that we have several parallel ports allowing the pipe to perform multiple accesses to the memory at the same time. For example, a pipe $M \longrightarrow M \longrightarrow M \longrightarrow M$ allows four parallel memory references resulting from four instructions "crossing" the pipeline.

The compute instruction contains the coding of the circuit representation of the operations and their arguments (as depicted in figure 1). Note that each argument of a pipe unit can be either a register or an internal stage. The number of bits in an instruction n depends on the amount of bits needed for this coding. We assume that the maximal number of pipeline units k per pipe is a known constant.

Move instructions- Constants and memory addresses are passed to a register via move instructions. The op field of the move instructions determines an operation on the target register, including: setting $R1 = const$, addition $R1+ = const$, ...

Branch instructions- Multiple branch instructions in the same VLIW are not allowed, and branch instructions are assumed to reside only in the instruction memory of the first pipe. In case of a branch instruction, the rest of the VLIW instruction should contain either a NOP or an extension of the branch (explained later). Branch instructions are divided two the following types:

Function call/ret- A function call is a branch instruction that saves the return address, and the caller registers in the individual register-stacks. There is a special register (not part of the 16 general set) that is used to save the return address. Function call advances the PC to the beginning of the function being called (the address part of the instruction). The return address is saved in the special register mentioned above (recall that this special register is also saved on its designated stack). The return-instruction restores the registers of the caller (except the register used to return a function value) and set the PC to the correct value. Recall that saving and restoring the register should be done in one parallel step in the IF-stage. Note that parameters are basically passed via registers only, though it is possible to use the main-memory and a "SP" register for that purpose too. This consumes one coding option from the op field of the branch instructions.

Loop instructions- The CPU supports only hardware loops, namely loops that are executed by loading a counter with the requested number of iterations and then executing it. It is possible to reset or modify the loop counter during execution but not to terminate it at the middle of an iteration. Though this complicates programming it saves the need to evict the pipeline when a branch instruction is executed. Note that when loop-end is encountered, determining the next value of the PC can be done immediately since it depends only on the current value of the loop's counter. Thus, the next instruction that is fetched is determined at the instruction-fetch (IF) stage which is the first stage of the pipe and there is no need to flash out instructions that have been wrongly fetched. We have two instructions:$loops\ counter\_register, loop\_end\_address$ to start the loop and $loope\ counter\_register, begining\_address$ to determine the loop-end. Loops instruction must check that the register counter is not zero and if so jumps after the loop's end address (otherwize the loop's counter is automaticaly decremented). Note that it is possible to use nested loops.

Conditional Jumps- are executed in the first stage of the pipe and therefore can affect the next instruction address (PC) without any need to flash instructions from the pipes. If the jump is evaluated to true the jump is a fall-through otherwise we set the PC to $PC = PC + offset$. Jumps contain a logical expression $(R_i\ rop\ R_j)\ lop\ (R_l\ rop\ R_k) \quad i < j < l < k$ that is used to evaluate the jump's condition where $rop = \{<, ==, >, \geq\}$ and $lop = \{AND, OR\}$. The reg_exp coding contains:

- An eight bits segment where the position of a bit set to one indicates the next register i, j, k, l participating in the reg_exp.
- A two bit segment indicates which half of the 16-registers i, j, k, l are referring too.
- A six bit segment for coding the three operators rop, lop of the $reg - exp$.

The offset can be either a register or a constant number (depending on the first bit). Since the VLIW instruction holds only one jump instruction at the first pipe, the rest of the VLIW space can be used to increase the $reg - exp$ of the first pipe to have an AND or OR of the other pipes' $reg - exp$s. (the exact formating of this option is determined by you).

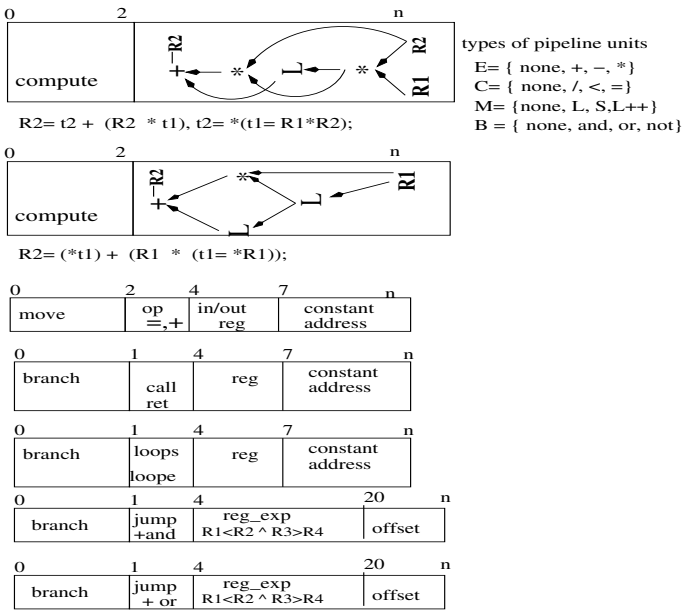

Figure 1: Instruction's format.

2 Schematics of the design

Figure 2 depicts the general structure of the tiny cpu. This CPU has two pipes $E \longrightarrow M \longrightarrow E$ and $M \longrightarrow E \longrightarrow M$.

- Each latch has multiple occurrences working as a shift registers so that older values can be used at a later stage.
- Memories can be made to work in one clock cycle, or better in two clock cycles including: placing the address and obtaining the value.
- The main memory is a multiport memory supporting parallel access of the $M \longrightarrow$ units of the pipes.

Figure 3 depicts the usage of a reconfigurable crossbar to implement the the different pipeline shapes and variable instructions of the tiny-cpu:

- It is recommended to use tri-state busses to implement the crossbar, however using muxes is also possible (but slower).
- The values in the latches connecting the pipeline stages can be used as inputs by several units simultaneously.
- Older values of the latches can be used for future stages.
- We assume that the connect/disconnect operation on the crossbar's busses occures during the positive edge of the clock while the registers/latches are updated at the negative edge of the clock. Thus at the same clock cyclevalues are loaded from the registers and latches and the output of the diffrent stages is also strored back in the registes and latches.

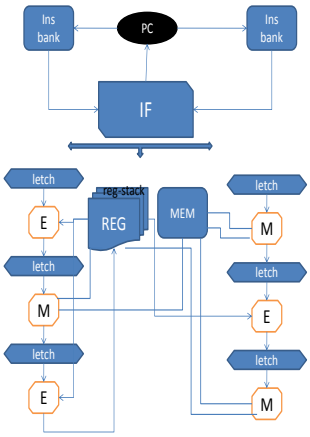

Figure 2: General schematic of the cpu.

- Registers are updated every clock cycle, thus if a register should maintain its current value the switches should connect its output to it sinput.

Figure 3 depicts a corssbar that implements two pipelines $E \longrightarrow M \longrightarrow E$ and $M \longrightarrow E \longrightarrow M$ with three registers $R1, R2, R3$. Figure 3 illustrates how the crossbar implements the parallel execution of two instructions $R1 = R1 + *(R1 + R2)$ and $R3 = (*R3 = R2 + *R2)$. The different stages of the parallel execution of the two instructions are illustrated as follows:

1. the $R1 + R2$ of $R1 = R1 + *(R1 + R2)$ and the $*R2$ of $R3 = (*R3 = R2 + *R2)$ are performed by setting all the swiches marked 1 (outputs are stored in latches $1/3$ respectively).

2. the $*(R1+R2)$ of $R1 = R1 + *(R1+R2)$ and the $R2+ *R2$ of $R3 = (*R3 = R2+ *R2)$ are performed by setting all the swiches marked 2 (outputs are stored in latches $2/4$ respectively).

3. the $R1 + *(R1 + R2)$ of $R1 = R1 + *(R1 + R2)$ and the $*R3 = R2 + *R2$ of $R3 = (*R3 = R2 + *R2)$ are performed by setting all the swiches marked 3 (outputs are stored in registers $R1/R3$ respectively).

An alternative to the use of the crossbar is that the state between pipeline stages includes.

1. Each pipe has a seperate set of leteches and no information is propagated between pipeline stages

2. Registers are updated only at the writeback stage which is common to all pipes.

3. Input registers are obtained sperately at the right stage of the pipe and use the current value of the register present in the register file.

4.
```
R3 = R2+R1+R1*R7;
[R4 = R3*R3 || R5 = R3+R3];
```

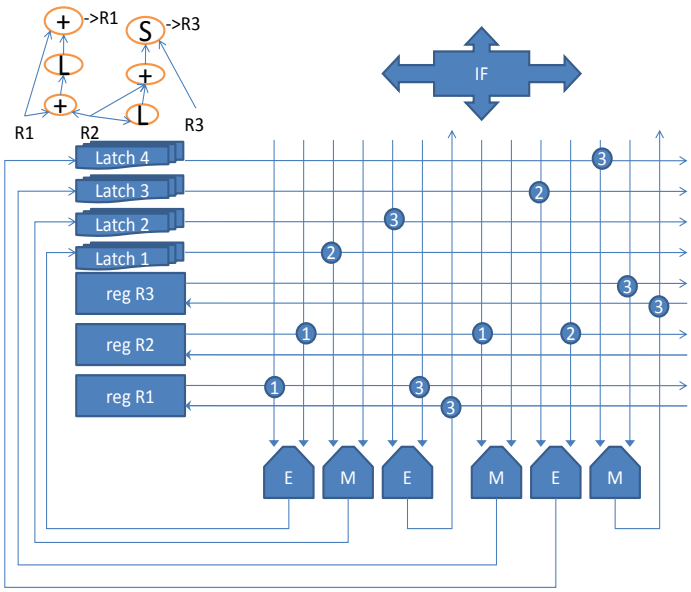

Figure 3: Using crossbar techniques to implement the Tiny CPU.

Since $R4 = R3 * R3$ is executed at the same pipeline as $R3 = R2 + R1 * R1 + *R7$ then it will observe the right updated value of $R3$s. However, $R5 = R3 + R3$ is executed in another pipe wherein the value of $R3$ is not updated.

5.
```
R3 = R2+R1;  /* R2+R3 is ready after the firs stage */
[R4 = R3*R3+R4*R6 || R5 = R4*R4*R3+R3];
```

3 Example

Following is the assembly code for the "part of transitive closure" calculation given in figure 4. Let's assume that the variable n is preloaded in the Data Memory Bank address 99. The input connectivity matrix is preloaded starting at address 100, row by row (i.e. $M[p][r]$ data is at the address $100 + p * n + r$). The following is pseudo-assembly using commands available in the Tiny CPU. Commands' arguments are described in our project definitions. This code is not optimal, it just depicts the assembly you would preload into the Instruction Memory banks. Comments are given given after each instruction while parallel instructions are separated by || (assuming that we have the suitable pipelines in the TCG.

```
MOVE R16 = 99  || R15 = 100
/* Moving the address of n and the start addr of M into registers*/
COMPUTE R0 = (*R16) || R1 = (*R16) || R2 = (*R16)
/* Loading the value of n into three counters' registers */
COMPUTE R16 = (*R16)
/* Loading the value of n into the same register */
```

```
LABEL0: LOOPS R0, LABEL1
/* Start of the i-loop, loop counter is R0 */
LABEL2: LOOPS R1, LABEL3
/* Start of the j-loop, loop counter is R1 */
LABEL4: LOOPS R2, LABEL5
/* Start of the j-loop, loop counter is R1 */
COMPUTE R3 = *(R15+ (R16-R0)*R16 + (R16-R2)) || R4 = *(R15+ (R16-R2)*R16 + (R16-R1))
/* Parallel computation of M[i][k] and M[k][j] values */
MOVE R5 = 0 || R6 = 0
/* Moving 0-constants into regs (preparation for Cond Jump) */
CONDJUMP (R3 > R5) AND (R4 > R6), offset=3
/* Conditional jump. If not true - jump offset=3 lines forward */
MOVE R5 = 1
COMPUTE R6 =  ((*((R16-R0)*R16 + (R16-R1)) = R5))
/* Store the value 1 into the address of M[i][j], and to register R6 */
* Decrement loop counters in parallel */
LABEL 5: LOOPE R2, LABEL4
LABEL 3: LOOPE R1, LABEL2
LABEL 1: LOOPE R0, LABEL0                                          /* Loop ends */
```

```
for(i=0;i<n;i++)
 for(j=0;j<n;j++)
  for(k=0;k<n;k++)
   if(M[i][k] > 0 &&
     M[k][j] > 0 ) M[i][j]=1
```

Figure 4: Example: part of a transitive closure computation.

4 Tasks

1. Create a code example in assembly for some simple algorithm that demonstrates:
   - Demonstrates the main features of loops and function calls.
   - Exploits the use of multiple and complex pipelines.
   - Each team should have a different example, so pls. get my confirmation before you select an example.
2. Write a C/C++ program that simulates the TCG and in particular its crossbar mechanism. This program must run on the example implemented in the previous item. This task should be completed until 15.11 or there will be a panelty of 4 points in the final grade.