University of Toronto

ECE 1769
Comprehensive Project

Final Report

Franjo Plavec

Toronto, April 2003

Introduction

The purpose of this project was to design a toy behavioral synthesis tool. The tool compiles input C program, and outputs the Verilog code that can be fed into a commercial logic synthesis tool. The final result is a circuit that performs the operation specified by the input C program.

The input C program is limited in a sense that it should not use pointers, dynamic memory allocation, aggregate data types, and function calls.

The project was divided into four phases (excluding the introductory phase of SimpMeta exploration) as follows. In phase 1, resource constrained scheduler was implemented. In the next two phases, register allocator, and functional unit binder were implemented. In the final phase, Verilog code generator was implemented.

Following sections present the implementation details of each phase, as well as the final results yielded by running the program on the small benchmark set.

Resource Constrained Scheduler

Resource constrained scheduler performs instruction (DAG) scheduling by employing list scheduling algorithm. List scheduling algorithm minimizes latency, under given resource constraints. List scheduling algorithm is a greedy algorithm. In every step of the algorithm, the DAG with maximum out degree is chosen.

Resource constraints are specified on the command line of the program. Along with the constraints specified on the command line, program assumes existence of "universal" library component, which performs all operations (opcodes) that are not specified on the command line. There is only one such component. If there are no constraints given, it is assumed that only one "universal" component exists.

Constants and labels are not scheduled, since there is no operation necessary for them. It is assumed that constant, and target address fetching is accounted in operations that use these DAGs as operands

All operations take exactly one cycle to execute. Program assumes non-SSA code as an input, and constructs its own precedence graph according to data dependencies.

Data dependence analysis:

Prior to scheduling DAGs, program performs data dependence analysis, in order to build precedence graph. Data dependence analysis takes into account all possible dependencies that occur in computations (i.e. RAW, WAR, and WAW dependencies). RAR dependencies are not taken into account, since they only occur in specific cases. Dependence analysis is done only locally (in basic blocks), and is based on the following algorithm:

1. Create an array of objects that represent dependence graph. Each object contains DAG, two lists (before and after), variable produces which determines what value DAG produces (if any), and a variable that determines if the DAG is 1st level DAG. DAG is called 1st level DAG if it doesn't have parent DAG (i.e. the ones that are returned by the getCodes method). Lists before, after, and variable produces are used as follows. List before contains indices (array indices) of DAGs that have to be scheduled before current DAG can be scheduled. List after contains indices of DAGs that can be scheduled only after current DAG has been scheduled. Each object also contains some other auxiliary variables.
2. Create an array with indices of 1st level DAGs
3. Traverse all DAGs, and fill lists before and after as a result of operation-operand relations, i. e. the operand has to be available in order to schedule the operation.
4. Traverse 1st level operands, and set variable produces. If any other DAG on the first level also produces that value, fill lists before and after appropriately to take care of WAW dependencies. Only 1st level DAGs are traversed, since only they can produce a value.
5. Traverse non-1st level DAGs. If a DAG with index k reads a value, try to find the two numbers i and j, such that i and j are indices of 1st level DAGs who both write the value that k reads, and $i<k<j$, such that i is maximum such number, and j is minimum such number. This basically means that i is the last DAG before k, that wrote the value k wants to read (thus taking care of RAW dependency), and j is the first DAG after k that writes the value (thus taking care of WAR dependency). If such numbers are found, lists before and after are filled appropriately.
6. At the end, check if the last 1st level DAG is a branch instruction. If so, it can only be scheduled after all other instructions have been scheduled, and list before and after need to be filled appropriately.

Register Allocator

Register allocator minimizes the number of registers used, while respecting the result of scheduling. Register allocator was implemented using the graph coloring algorithm. Interference graph is produced by employing global data-flow analysis.

It is assumed that branch operations do not require a register, since they do not produce any value. Also OP_STR operations do not require register, since they store values to the memory (or some sort of I/O).

Steps of the algorithm (method global):

1. The scheduler is called for each block (method schedule).
2. Data-flow analysis uses some information produced by scheduler. The data produced by scheduler is adapted by removing dependencies related

to the arrays (method remove_dep), since array references do not incur limitations on registers (i. e. storing and loading is done to/from memory). This only removes the data dependence resulting from the array references, not the allocation of register for the load operations.

3. Local data-flow analysis is performed (method local). Local data flow analysis fills alive set with the information that can be extracted from the relations between the DAGs in a single block. It also produces uses, defsout, and killed sets, which are described in more details in Data-flow analysis below.
4. After all blocks have been scheduled, and local analysis performed, iterative algorithm is called (method iterate) to produce reaches set, which is described in more details in Data-flow analysis below.
5. Based on the information in the uses and reaches sets, alive set is updated accordingly (method update_alive).
6. Interference graph is produced from the information in the alive set (method interference) Two DAGs interfere with each other if they are both alive in the same cycle.
7. Since some DAGs need to share the same register, interference graph is updated accordingly (method adjust).
8. Coloring algorithm is employed to perform register binding (method regbind). The standard graph coloring algorithm was used. Vertex elimination scheme based on Less-flexible-first principle was used (method sigma).
9. Changes made in step 7 are reversed, so all DAGs have assigned registers (method readjust).
10. Finally, the number of registers required is returned.

Data-flow analysis:

Data-flow analysis was based on the algorithm found in [1]. Program is first analyzed locally in each basic block. This analysis produces local alive set, as well as sets uses, defsout, and killed.

Local alive set contains, for each cycle, list of variables that need to be alive in that cycle due to operand - operation relations inside the basic block.

Uses set contains the list of all DAGs that read values within the block that have no prior definitions within the block. Note that this can only happen if DAG reads some variable, since operand - operation relations do not cross basic block boundaries.

Defsout set contains the list of all DAGs within block b that reach the end of the block. Only definitions of the variables are in this set, since no DAG in other basic block will ever use DAGs from this basic block as an operand.

Killed set contains the DAGs from the block that produce some variable. The original definition requires the killed set to contain all variables that are killed. We store actual DAGs in the set, since the information on actual variable name can be extracted from the DAG that produces the value.

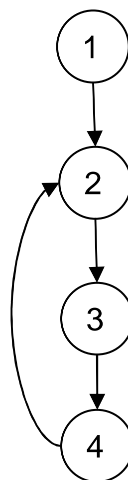
Sets uses, defsout, and killed are used in an iterative algorithm that produces the set reaches. Reaches set contains a list of all DAGs whose definitions can reach the block. For some block b, and its predecessor p:

for each p:

$$reaches(b) = reaches(b) \mathbf{U} \{defsout(p) \mathbf{U} (reaches(p) \cap \neg killed(p))\}$$

This basically means that the block can be reached by any definitions that reach the end of all of his predecessors, including those that reach the predecessor, and are not killed in it. In order to calculate reaches, iterative algorithm is employed. Algorithm starts with the assumption that all reaches sets are initially empty. Algorithm recalculates reaches sets, until the stable solution is reached.

Having the reaches and uses sets, local alive set can be updated with global information on alive DAGs. For any given block b, every variable in set uses is paired with all definitions of that variable in reaches set. For each such pair, the DAG that produces the value is added to the set alive for all cycles between the definition and the use of the variable. Since this can include multiple basic blocks, all possible paths from the DAG that defines the variable, to the DAG that uses it, have to be found. This is performed by traversing the control flow graph recursively from the definition to the use of the variable. If the use of the variable is reached, all the blocks on the path have to have that DAG alive during all the cycles. Special care must be taken to handle cycles properly. Every basic block visited is marked, and the mark is removed when returning from the recursion to enable two paths to go through the same basic block. When the use of the variable is reached, the same algorithm is employed once again, to check if the definition can reach the use through some cycle. Graph 1 shows such an example.



Graph 1. Handling cycles in a control-flow graph

If basic block 1 defines a variable which is used in basic block 2, it also needs to be alive in blocks 3 and 4. This is due to the back-edge from basic block 4 to the basic block 2, which can cause the definition from the block 1 to reach the use in block 2 through basic blocks 3 and 4. The prerequisite is, that blocks 3 and 4 do not kill that variable. This problem is simply solved by recursively calling the algorithm with the basic block 2 as the first and the last node in the path sought.

Extracting the interference graph from the alive set is straightforward. If two DAGs are alive in the same cycle, they interfere with each other. However, some care must be taken of two definitions that reach the same use of the variable. These definitions have to be stored to the same register. However, only one of the definitions can be alive at a time. This is handled by marking all such cases (called "shared" DAGs), and using only one of the shared DAGs in the interference graph. When the graph coloring algorithm assigns registers, the same register is used for all DAGs in the shared list.

Functional Unit Binder

Functional unit binder minimizes the cost of bus drivers, while respecting the results of scheduling and register allocation. Functional unit binding is implemented using weighted clique partitioning algorithm.

Method fu constructs the compatibility graph. Two DAGs are compatible if they are not scheduled at the same cycle. Each compatibility edge has affinity (called weights in the code, but they are really affinities) equal to the sum of numbers of common sources and common destinations. Affinities are used for weighted clique partitioning algorithm (method clique).

In some cases, the weighted clique partitioning algorithm produces more cliques than available functional units of certain type. This is because the greedy algorithm is used, which always selects the edge with greatest affinity. This can result in sub-optimal number of cliques. The problem is fixed during final functional unit binding. Each DAG in a clique that can not be scheduled, because there are fewer units than cliques, is assigned to an appropriate clique already bound. This is always possible, since instruction scheduling will never schedule more instructions than there are functional units available.

Clique partitioning algorithm is not applied if there is only one functional unit available for each type of the operation, since all DAGs have to share the same functional unit.

Verilog Code Generator

Verilog code generator is divided into component, datapath, controller, and top-project generator. The generator produces files comps.v, datap.v, control.v, and top.v, which can be fed to the commercial logic synthesis tools.

Component generator

Component generator generates the functional units that perform operations specified one the command line. It also generates multiplexers.

Not all of the opcodes supported in SimpMeta are supported in component generator. Only opcodes necessary to test a couple of simple programs were actually implemented. Other opcodes could be implemented in a similar fashion, although some operations could introduce additional problems. For instance, multiplication requires more than one cycle for the operation. This could be handled by stalling all other operations, until the multi-cycle operation is finished. This is the simplest possible solution, but it is far from optimal.

Component generator does not use standard library components (except for the lpm_ram_dp memory module). This approach allows virtually any combination of the opcodes performed by a single functional unit. Standard components library could be used if such a feature is not needed. If the functional unit performs more than one operation, it has func input, which selects the operation that should actually be performed.

The only operations that should be performed by a single functional unit are OP_LOD and OP_STR, since only single-port memory is used. The functional unit that performs OP_LOD and OP_STR operations also instantiates the memory module. This is fine, since it is the only functional unit that requires access to the memory.

The size of the memory is determined by the total size of the arrays declared in the input C program times four. The size of the memory is four times larger than necessary, since the intermediate code generated by SimpMeta assumes byte-addressable memory. For the simplicity, word-addressable memory four times larger is instantiated, and only every fourth word is actually used. Currently, only global arrays are supported, although local arrays could be handled in the same way as the global arrays.

Component generator doesn’t use library component for a multiplexer, since versions of Verilog available do not fully support two-dimensional ports. Custom multiplexers are generated instead, with the number of inputs, and the size of select inputs as required.

Datapath Generator

Datapath generator generates the Verilog code that instantiates and interconnects the components of the datapath: registers, functional units and multiplexers.

Registers are implemented by using the standard library component lpm_ff. Multiplexers to registers and functional units are connected according to the results of the register allocation and functional unit binding. Appropriate number of functional units of certain type is instantiated, as specified on the command line.

Controller Generator

Controller generator generates Moore type finite state machine that controls the functional units, multiplexers, and registers.

The number of states required is the sum of number of cycles required for each basic block (as determined by scheduling algorithm). Each DAG is assigned the state according to the scheduling, and the possible set of next states is determined by the control flow (i. e. possible branch targets). Two artificial states are added to the machine. START_STATE resets all the registers to zero, while END_STATE indicates that the machine has finished with the program execution.

In each state, the set of the control signals are activated, according to the given schedule. More precisely, functional units that perform operation specified by DAGs scheduled in given cycle/state have to be enabled, and appropriate function selected. Select inputs to the multiplexers have to be set according to the operands of a given DAG. If the DAG produces certain value, appropriate register needs to be enabled, and the select input of the multiplexer that feeds the register has to be set to select the appropriate functional unit.

Top-project Generator

Top project generator produces the Verilog file top.v which instantiates control and datapath modules, and interconnects them appropriately. This module is used as a top level module in a circuit simulation.

Results

The functionality of the program was tested by running the program on a small benchmark set of six programs: adding first N integers, multiplication by addition, division by subtraction, simple sorting algorithm, array reversal, and matrix multiplication.

Results of the synthesis were fed to the ModelSim ALTERA 5.6a, a commercial tool for Verilog simulation [2].

Memory, when used, was initialized using Intel hex-format files. This approach enables testing the programs for various input data without the need for re-compilation. The results of the simulation are shown in the following subsections.

Adding First N Integers

The program simply traverses given number of integers, and sums them.

```
//Adds firs 50 integers
void main () {

    int i, a;

    a = 0;
    for (i = 1; i <= 50; i++)
            a += i;
}
```

The results of the simulation of the Verilog code produced for N = 50, are shown in the Figure 1.

Figure 1. Results of adding first 50 integers

Figure 1 shows a couple of final cycles/states of the program execution. First two signals on the top show current and the next state of the state machine. After the machine enters the state number ten, it stays in that state indefinitely, since it is the END_STATE. At that point register 4 contains the result of the program execution, which is 1275. The figure also shows the values of the result in a couple of previous steps, as well as some other signals.

Multiplication by Addition

The program multiplies two numbers a and b by adding the number b a times. The program chooses larger number for addition, to reduce the number of steps required.

```
//Multiplication by addition
void main () {

    int a = 31, b = 17; //operands
    int i, x, y, res = 0;

    //select smaller one, to reduce number of steps
    x = (a <= b) ? a : b;
    y = (a <= b) ? b : a;
    for (i = 0; i < x; i++)
            res += y;
}
```

The results of the simulation of the Verilog code produced for a = 31, and b = 17, are shown in the Figure 2.

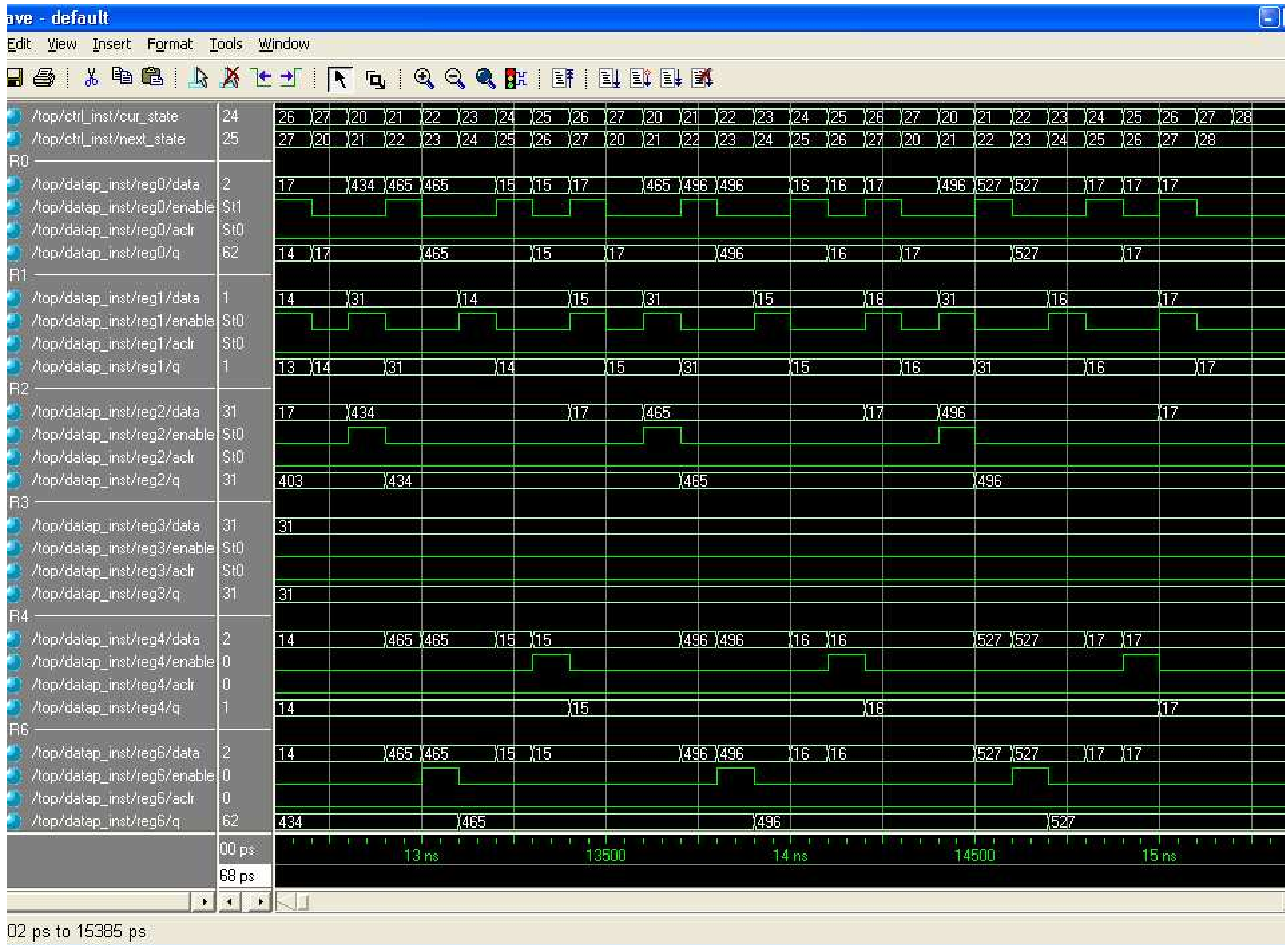


Figure 2. Results of multiplying 31 by 17

The result of the multiplication is placed into the register R6. The couple of final steps are also shown (434 + 31 = 465, +31 = 496, +31 = 527 = 17 * 31). After entering the state 28, machine stays in that state indefinitely.

Division by Subtraction

The program divides two number by subtracting the divisor from dividend as long as the dividend is larger than the divisor, and counting the number of subtractions.

```
//Division by subtraction
void main () {

    int a, b, res;

    a = 82;
    b = 7;

    res = 0;
    while (a > b) {
        a -= b;
        res++;
    }
}
```

The results of the simulation of the Verilog code produced for dividing 82 by 7 are shown in the Figure 3.

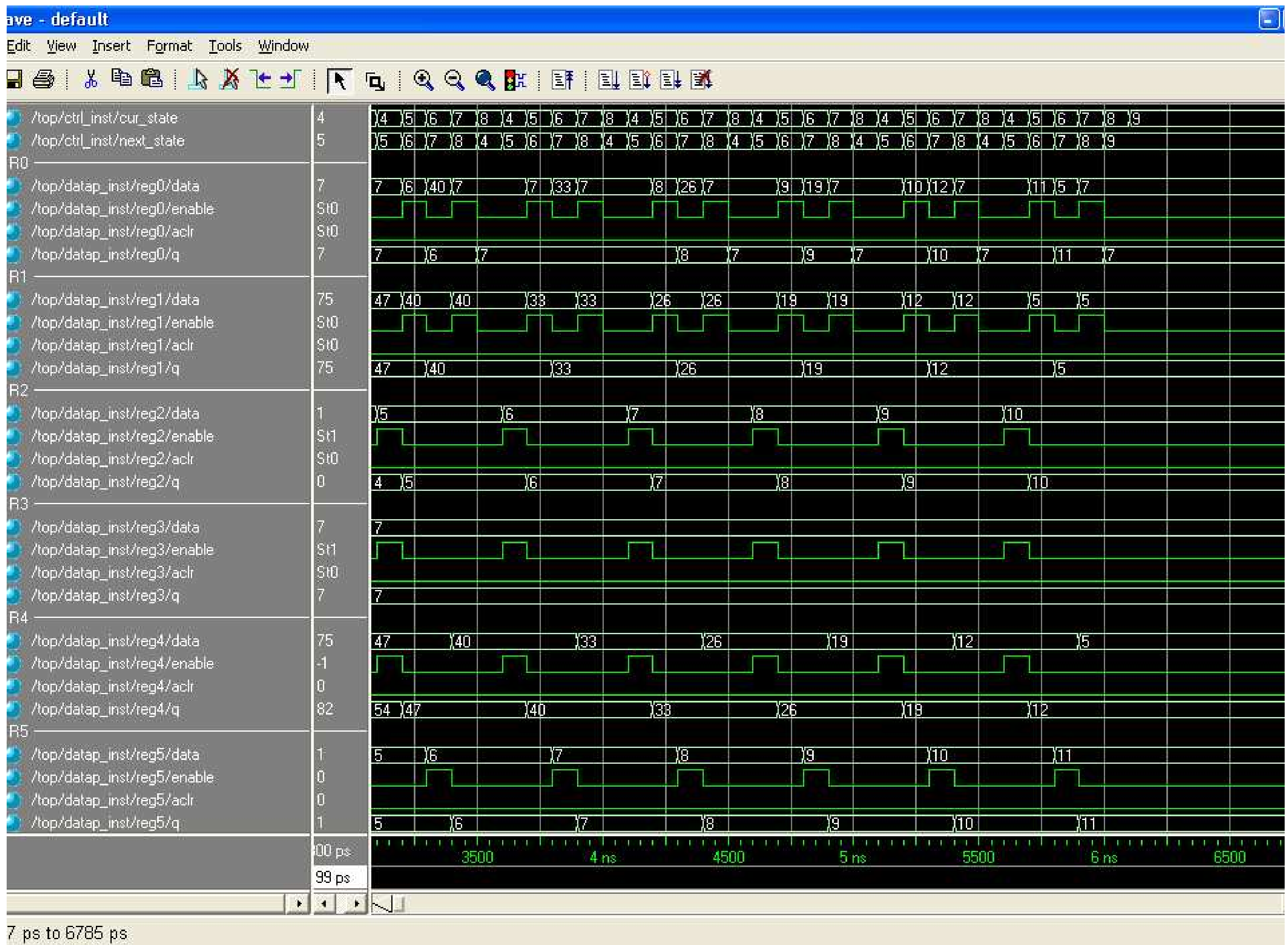


Figure 3. Results of dividing 82 by 7

The result of the division is placed into the register R5. The couple of final steps are also shown. After entering the state 9, machine stays in that state indefinitely.

Simple Sorting Algorithm

The program compares every element with the smallest element in the rest of the array, and puts the smaller one to the current position.

```
//Simple sort algorithm
int a [10];

void main () {

    int i, j, min, min_index, temp;

/*  a[0]=7; a[1]=9; a[2]=2; a[3]=58; a[4]=32;
    a[5]=234; a[6]=1; a[7]=100; a[8]=512; a[9]=17;
*/
    for (i = 0; i < 10; i++) {
            min = 2147483647;
            for (j = i; j < 10; j++) {
                    if (a [j] <= min) {
                            min = a [j];
                            min_index = j;
                    }
            }
            temp = a [min_index];
            a [min_index] = a [i];
            a [i] = temp;
    }
}
```

The results of the simulation of the Verilog code produced for sorting the array [7, 9, 2, 58, 32, 234, 1, 100, 512, 17] are shown in the Figure 4.

Figure 4. Result of an array sorting

Unlike previous programs, this program uses arrays. Figure 4 shows the sorted array, as a result of the simulation of the Verilog code. Middle (grey) column shows the original array order. Note that every fourth location is used, as explained in the Component Generator subsection. The array was initialized using the hex-file memdata.hex.

Array Reversal

The program reads every element of the array, and writes it to the another array in the reversed order.

```
//Reversing one array and storing it to another
int a [10], b[10];

void main () {

    int i;

//  a[0]=7; a[1]=9; a[2]=2; a[3]=58; a[4]=32;
//  a[5]=234; a[6]=1; a[7]=100; a[8]=512; a[9]=17;

    for (i = 0; i < 10; i++)
            b [i] = a [9-i];
}
```

The results of the simulation of the Verilog code produced for reversing the array [7, 9, 2, 58, 32, 234, 1, 100, 512, 17] are shown in the Figure 5.

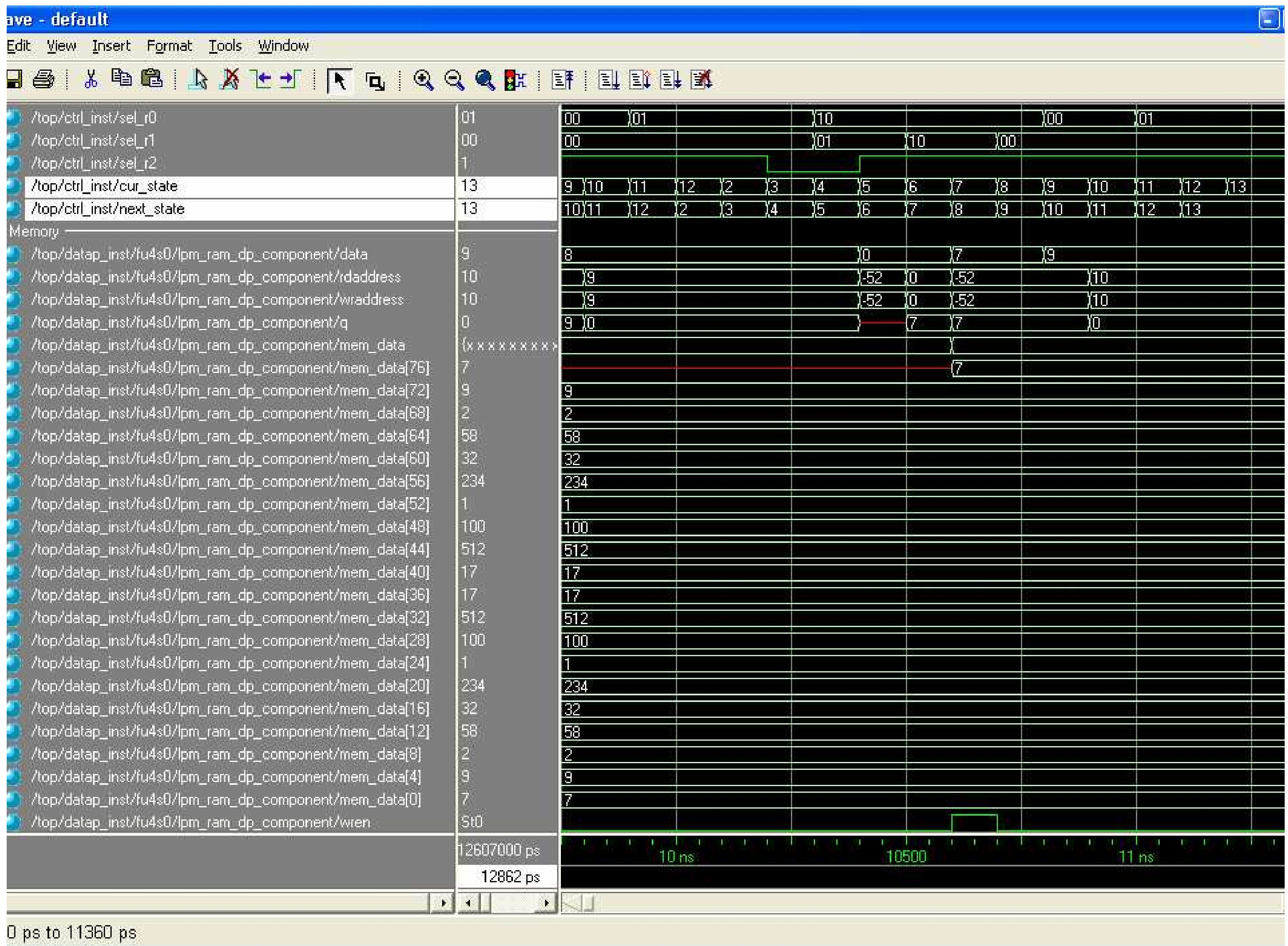


Figure 5. Results of the array reversal

Figure 5 shows a couple of last states/cycles of the reversal algorithm. Both original (indices 0 to 36), and the reversed (indices 40 to 76) arrays are shown. In the state 7, last number is written to the memory by activating memory's wren signal (bottom most signal). After the machine enters the state 13, it stays in that state indefinitely.

Matrix Multiplication

The program performs simple matrix multiplication algorithm. Matrices a and b are multiplied, and the result is stored in a matrix c. Element multiplication is performed by addition, as described in Multiplication by Addition subsection.

```
//Matrix multiplication
int a[3][2], b[2][4], c[3][4];

void main () {

    int i, j, k;

            int u, v, l; //operands
      int x, y, res;

    goto L2;
    //Multiplication is implemented by addition
    //select smaller one, to reduce number of steps
L1: x = (u <= v) ? u : v;
    y = (u <= v) ? v : u;
    res = 0;
    for (l = 0; l < x; l++)
            res += y;

    goto L3;

L2:
//  a[0][0] = 1; a[0][1]= 2;
//  a[1][0] = 3; a[1][1] = 4;
//  a[2][0] = 5; a[2][1] = 6;

//  b[0][0] = 2; b[0][1] = 4; b[0][2] = 6; b[0][3] = 8;
//  b[1][0] = 1; b[1][1] = 3; b[1][2] = 5; b[1][3] = 7;

    for (i = 0; i < 3; i++) {
            for (j = 0; j < 4; j++) {
                    c [i][j] = 0;
                    for (k = 0; k < 2; k++) {
                            u = a[i][k];
                            v = b[k][j];
                            goto L1;
L3:
                            c[i][j] = c[i][j] + res;
                    }}}}
```

Two matrices being multiplied are:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} * \begin{bmatrix} 2 & 4 & 6 & 8 \\ 1 & 3 & 5 & 7 \end{bmatrix} = \begin{bmatrix} 4 & 10 & 16 & 22 \\ 10 & 24 & 38 & 52 \\ 16 & 38 & 60 & 82 \end{bmatrix}$$

Number multiplication could have been inlined in the loop. However, this way is more challenging for the algorithms used, since it results in more complicated control-flow graph.

The results of the simulation of the Verilog code produced for multiplying matrices are shown in the Figure 6.

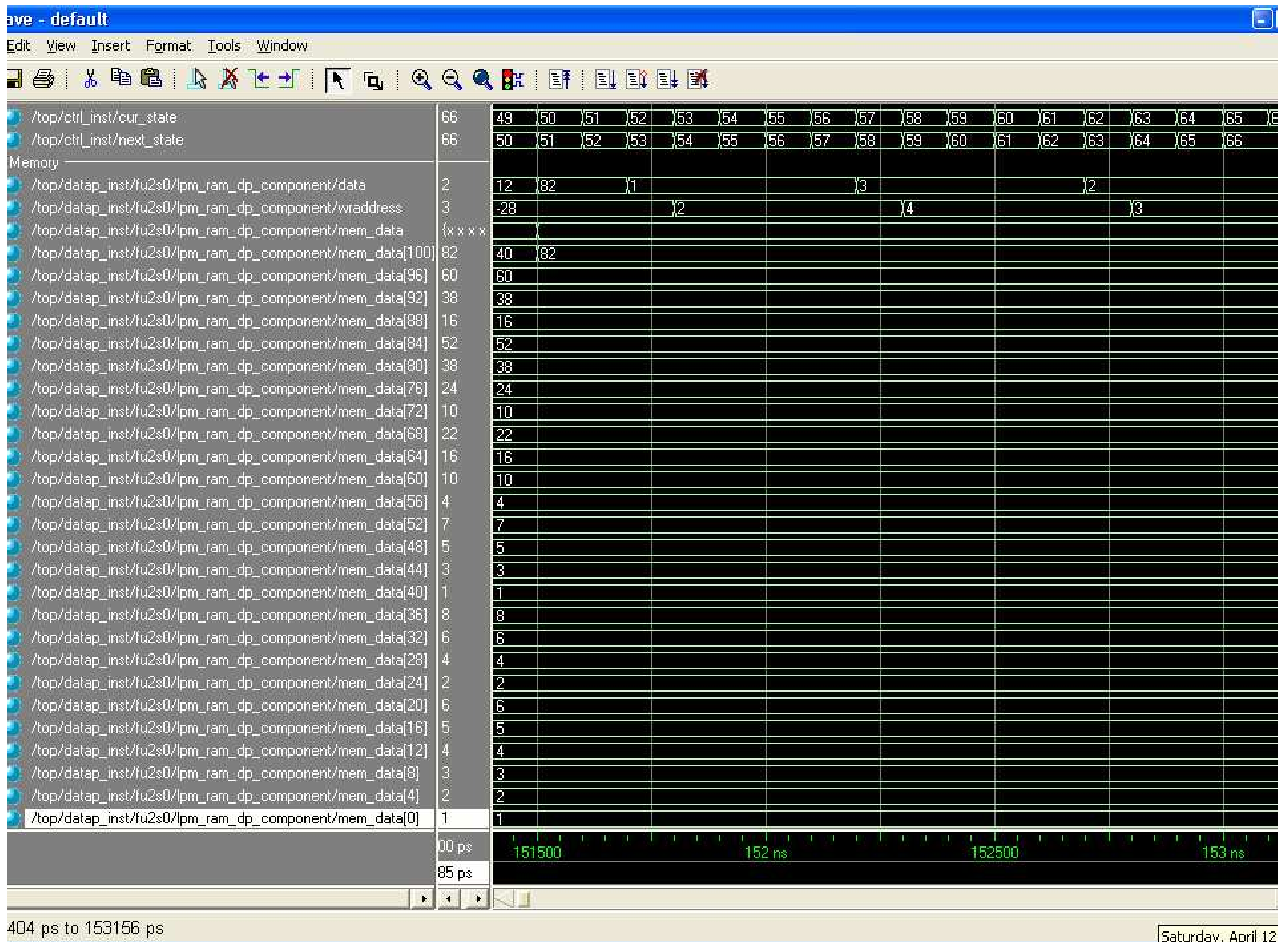


Figure 6. Results of the matrix multiplication

Figure 6 shows a couple of last states/cycles of the matrix multiplication algorithm. Contents of matrices a (indices 0 to 20), b (indices 24 to 52), and resulting matrix c (indices 56 to 100) is shown. In the state 50, last part of the calculation is written to the memory (5*8 + 6*7 = 82). After the machine enters the state 66, it stays in that state indefinitely.

Conclusion

The final result, the toy behavioral synthesis tool, showed to be successful in generating correct output for small test programs. Although current version lacks lots of possible functionality, most of that functionality could be added incrementally, in the same, or similar way the existing functionality was implemented. However, some more advanced features, like dynamic memory allocation, or function calls, would require significant changes in the structure of the tool.

The project has shown to be an invaluable experience. Although only a toy tool, it has provided a good insight in the structure of the behavioral synthesis tools.

References

[1] R. Allen and K. Kennedy: Optimizing Compilers for Modern Architectures: A Dependence-Based Approach, Morgan Kaufmann Publishers, 2002.

[2] Model Technology, ModelSim Documentation, http://www.model.com/support/documentation.asp, April, 2003