

ECE1769

Behavioral Synthesis Tool
April 22, 2002

Navid Azizi
984301910

1 Introduction	3
2 Design	3
2.1 Conversion from SimpMeta to Shared Data Structures	3
2.1.1 Data Structures	3
2.1.2 Conversion Process	4
2.2 Instruction Scheduling	5
2.2.1 Scheduling Heuristics	6
2.3 Register Allocation	7
2.4 Functional Unit Binding	8
2.5 VHDL Exporting	9
2.6 Summary of Functionality	10
3 Results	10
4 Conclusion	11
A1 Appendix 1: Testbenches	12

1 Introduction

Synthesis involves the automatic mapping of an abstract functional description of a system onto a physical structure. The synthesis process can be subdivided into a sequence of smaller mappings; these include synthesis at the behavioral level (algorithm level), register transfer level and logic level. This report summarizes the workings of a developed tool that performs synthesis at the behavioral level, mapping a C program into structural VHDL. Section 2 discusses the methodology and design of this tool and is followed by section 3, which presents the results on a set of benchmarks. Finally, section 4 concludes this report.

2 Design

The design of the behavioral synthesis tool involves many stages: scheduling of operations, register allocation, functional binding and the creation of datapath and control modules. The design of these operations was accomplished by separating the full design into four modules. Information between the modules was transferred through a set of data structures.

Using the auxiliary program, `mcc`, C programs were first mapped onto an intermediate format consisting of directed acyclic graphs (DAG) for each basic block (BB). This representation was used as input for the synthesis tool. The first step involved in the synthesis tool was to create the shared data structure that could be used by all other modules. This first step was then followed by scheduling of operations (DAG nodes), register allocation, functional unit binding, and finally creation of the control and datapath in VHDL. The following subsections will describe each module, and the algorithms used within them in detail.

2.1 Conversion from SimpMeta to Shared Data Structures

In order to improve the ability to traverse and search data, a decision was made to convert the SimpMeta representation to a new set of shared data structures. The main structures used within the conversion will be explained, and then the associated transfer of information will be covered.

2.1.1 Data Structures

Figure 1 shows the `t_blocks` type which contains pointers to all BB structures which are stored within `t_block` datatypes. Each BB is assigned an implicit ID which is used as an offset to the blocks pointer to arrive at the needed block.

All information regarding a BB is placed within the `t_block` datatype, which is shown in Figure 2. `t_block` contains the predecessors and successors of the BB (which are stored in integer vectors (`ivec`)); pointers to the nodes (DAG nodes) within the block; and the live sets used during register allocation. While the `ivecs` used for the predecessor and successor blocks holds the IDs of other BBs, the `ivecs` used for `live_set_in` and `live_set_out` hold the IDs of variables.

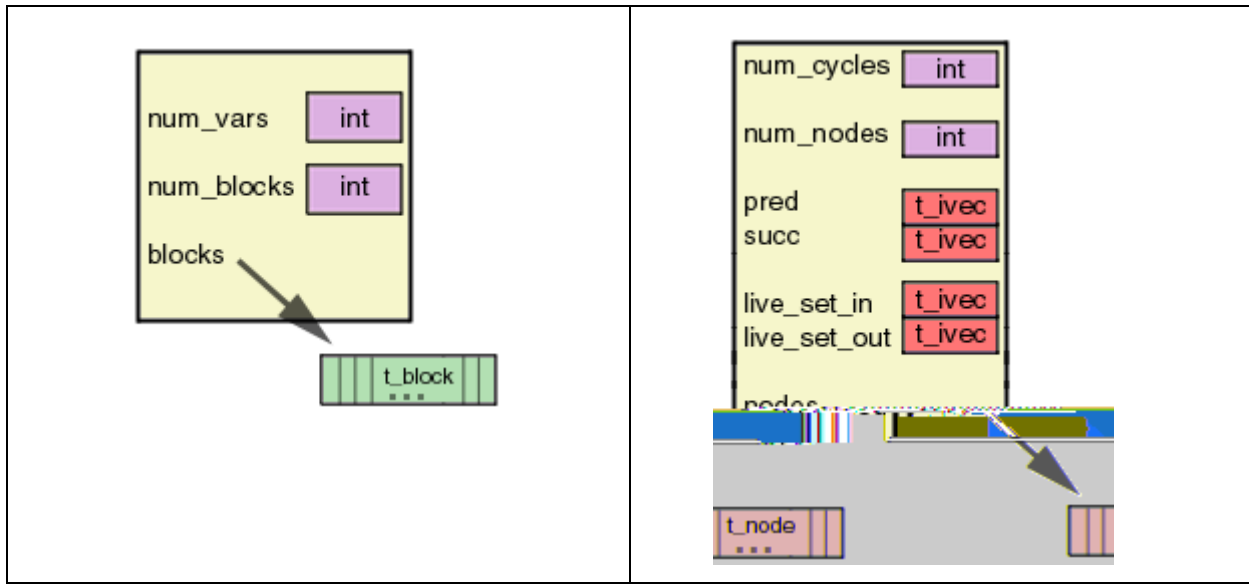


Figure 1: t_blocks Data Structure

Figure 2: t_block Data Structure

The `t_node` structure, shown in Figure 3, holds information relevant to a DAG node, including the `dag_id`, the operation, the sources and targets of the instruction and the immediate successors and predecessor nodes (which are found by analyzing dependencies). The `t_node` structure also holds scheduling information through `min_cycle`, `max_cycle` and `cycle` and the functional binding information through `func_unit`.

A final global data structure is the `t_assoc` datatype, shown in Figure 4. This structure maps the variable IDs used within the rest of the code back to a human readable format. For parameters and variables the mapping is from IDs to names, and for temporary stored values the mapping is from variable IDs to the ID of the DAG node that produces the value..

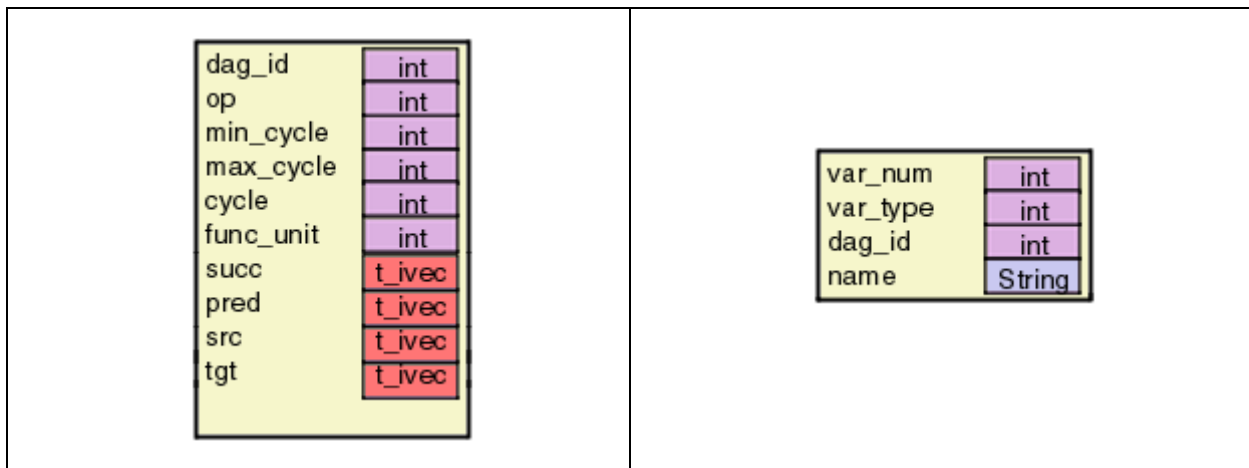


Figure 3: t_node Data Structure

Figure 4: t_assoc Data Structure

2.1.2 Conversion Process

The functions in `list.[ch]` perform the conversion from `SimpMeta` to the data structures described above. The hook into the conversion process is:

```
t_blocks alloc_and_load_blocks(SimpMember member)
```

which takes a `SimpMeta SimpMember`, which is the function contained in the C program, and returns a `t_block`. During the process all DAGs that are functional or branch DAGs are placed in the new data structures. Most loads and stores are not considered, except to retrieve the target and source operands of the functional and branch DAGs which they surround. The reasoning behind this is that loads and stores become register reads and writes which are not operations in themselves.

There is, however, an exception to the ignoring of loads and stores. Take for an example a statement such as `a = b`. This operation, while not taking up any useful time, creates dependencies between other DAGs and thus is included as one of the nodes within the `t_block` structure so that correct scheduling can be performed. There are also other special cases where other types of operations are included during the conversion process.

2.2 Instruction Scheduling

The goal of the instruction scheduling portion of the tool was to implement a resource constrained scheduler, using a list scheduling algorithm. More specifically, this portion of the code was to take in a number of functional units and then assign each DAG within a BB a cycle number such that the cycle count for the BB would be minimized.

The algorithm implemented followed the general structure shown in Figure 5.

```
for (iblock = 0; iblock < blocks.num_blocks; iblock++){  
    find_dependencies(blocks.blocks[iblock]);  
    end.min_cycle = asap(blocks.blocks[iblock], start);  
    end.max_cycle = guess_total_time(end.min_cycle, blocks.blocks[iblock], resource);  
    start.max_cycle = alap(blocks.blocks[iblock], end, end.max_cycle);  
    list_scheduling_with_resource_constraint(blocks.blocks[iblock], start, end);  
}
```

Figure 5: Outline of Scheduling Algorithm

For every BB the dependencies between all nodes within the BB is found, and then an “As Soon As Possible” (ASAP) algorithm creates an instruction scheduling that only depends on instruction dependencies (and not on any resource constraints). The ASAP algorithm sets the `min_cycle` fields within the nodes, and also sets the minimum time for which the BB could be complete.

Following the ASAP algorithm, an “As Late As Possible” (ALAP) algorithm needs to be run to determine the `max_cycle` time for all nodes. The ALAP algorithm, in addition to the node dependency information, needs the total cycle time for the BB. This number, however, will not be known until the complete scheduling algorithm is complete. Thus, a heuristic is needed to guess the total time given the resource constraints. More on this heuristic will follow in the next section.

Once the maximum number of cycles for the BB is guessed through the heuristic, the ALAP algorithm sets the `max_cycle` times for all nodes. At this time the minimum and maximum cycle time for all DAGs within the BB have been set and the list scheduling algorithm can be run.

The list scheduling algorithm starts with a list of candidate DAG nodes, which do not have any dependencies. The algorithm tries to schedule as many operations as possible given the resource constraints. When the number of candidate DAG nodes outnumbers the number of functional units, a selection process must occur to select specific DAG nodes over others. An explanation of this selection process will follow in the next section.

Once all functional units are busy, a new list of candidate DAG nodes is created by taking into consideration the results which have become available by the scheduling of operations in the previous cycle. This iterative process of finding candidates, selecting individual DAG nodes, and recomputing candidates is performed until all DAG nodes have been scheduled.

2.2.1 Scheduling Heuristics

The scheduling algorithm is more or less straightforward except for the creation of two heuristics: one for guessing the total time required for the computation of the BB, and another for selecting nodes to schedule if there is resource contention. The selection heuristic is explained below followed by the guessing heuristic.

A selection heuristic is needed when two or more DAG nodes can be scheduled at one time, but resources are not available to schedule all candidate DAG nodes. The created heuristic involves giving each candidate a score and then selecting the candidate with the largest score. The score used to select nodes is shown in Figure 6. The score depends on three properties of the candidate:

1. Out Degree
2. Mobility ($\text{max_cycle} - \text{min_cycle}$)
3. Max_cycle

```
Score = OUT_DEGREE_CONSTANT*candidate.out_degree;
Score += MOBILITY_CONSTANT/candidate.mobility;

if (candidate.max_cycle > cycle)
    Score += TIMESTEP_PENALTY*(cycle - candidate.max_cycle + 1);
else
    Score += TIMESTEP_CONSTANT/(candidate.max_cycle - cycle);
```

Figure 6: Score Function to select candidates

The rationale behind using these three properties is as follows:

1. Out Degree: Nodes with larger out degrees have more instructions depending on the results produced by the node. Scheduling nodes with larger out degrees allows for more candidates to be available in the next cycle in turn giving the list-scheduling algorithm more freedom to produce a good schedule.
2. Mobility : Nodes with larger mobilities have more freedom to move within the schedule before their scheduling time affects the scheduling of other nodes. These nodes are given smaller importance.
3. Max_cycle: If a node's max_cycle time, as determined by the ALAP algorithm, is larger than the current cycle time then it is known that the node is within the critical path of

dependencies and it will extend the total cycle time of the BB. Therefore a large penalty is given to the node in proportion to the number of cycles the scheduler has surpassed its `max_cycle` time. If the `max_cycle` time is still less than the current cycle then nodes with larger `max_cycle` times are more flexible and thus are given smaller scores.

The selection of constants and penalties that complete the heuristic can be largely arbitrary, and a sweep of the parameters among many benchmarks is needed to determine the best constants.

The heuristic to guess the total BB cycle time, which is needed for ALAP, consists of the following formula:

$$factor = \left(\sum_{resources} \frac{(need_i + 1)^2}{(have_i + 1)^3} \frac{need_i}{num_nodes} \right) + 1$$
$$guess = \min(\text{avg}(factor, min_time), min_time)$$

This heuristic traverses over all resources and by using the number of resources needed for the BB computation, $need_i$, and the number of resources available, $have_i$, computes a guess. The intuition behind the formula is twofold

1. The number of resources needed, $need_i$, over the number of resources that are available, $have_i$, is proportional to the number of times that a stall can occur due to the unavailability of resources. $have_i$ is cubed instead of squared to account for the fact that many instructions have dependencies that disallow them from occurring at the same cycle. The addition of 1 reduces the large increase in the factor when $have_i$ is close to 1.
2. The proportion of instructions within the BB that are of a specific resource impacts on the possibility that a stall may occur. For example, if there are many of instructions that need a resource, but the instructions only occupy 25% of the instruction count within the BB, then the possibility of other instructions causing dependencies within the code is high, lowering the possibility of a stall.

2.3 Register Allocation

Register allocation involves determining the minimum number of registers needed to store the values used within the execution of the C function. Analysis must be done over all BBs to determine the interval graph of a variable. Then variables that do not overlap or interfere with each other can share the same register.

Since the analysis of liveness of different variables must occur over all BBs, which can include branches and loops, a simple left-edge algorithm was not implemented. An overview of the algorithm used is shown in Figure 7. It involves performing a live-set analysis to create an interference graph between all variables. The interference graphs is used to order the vertices for a coloring algorithm, which allocates registers for each variable.

The live-set analysis creates the interference graph for the C function. The analysis steps through each scheduled instruction in post-order, creating an edge between the sources and

targets of the instruction and all variables currently in the live set. Finally the live-set analysis removes the target of the instruction from the live set and adds the sources to the live set before proceeding to the next instruction in the BB.

```
register_interference_graph = live_analysis(blocks);
order = order_vertices(register_interference_graph);
colors = color_vertices(register_interference_graph, order);
```

Figure 7: Overview of register allocation algorithm

To ensure the correctness of the live-sets and the interference graph in the presence of loops, the live-set analysis has to occur twice: first to initialize the live-sets at the boundary of BBs and second to recalculate the live-sets and set the edges in the interference graph.

After the interference graph is computed, the ordering of vertices for coloring is determined. The vertex orders are determined by selecting the vertex with the minimum degree and removing it from the graph. This step is performed iteratively until all vertices are removed from the graph. Finally with the graph empty, the order with which the vertices were removed is reversed and used to color the nodes.

The reason for using the ordering explained above is that nodes with larger flexibility have smaller degrees and are colored last. The coloring algorithm used is a simple greedy coloring algorithm. Vertices are colored with the first color that is different to all adjacent colors.

2.4 Functional Unit Binding

Functional unit binding assigns each operation to a specific functional unit. Appropriate functional unit binding can lead to smaller interconnection networks between the registers and functional units.

```
for (iresource = 0; iresource < max_resources; iresource++) {
    graph = create_compatibility_graph(blocks);
    While (num_nodes > num_functional_units) {
        select_two_nodes_and_combine(graph);
    }
}
```

Figure 8: Functional Unit Binding Summary

The algorithm used for functional unit binding is summarized in Figure 8. For each type of resource available within the program, a compatibility graph is created. The nodes within the compatibility graph include all nodes in all BBs that use the specific functional resource. Edges exist between nodes if and only if the two nodes are not scheduled at the same cycle in the same BB. Moreover, each edge is given a weight. The weight given to each edge is the sum of shared register inputs and register outputs. For example take the two instructions $a = b + c$ and $d = e + f$, and assume that after register allocation a and d have been allocated the same register. Furthermore, b and e have been allocated the same register, but c and f are allocated different registers. These two nodes would then have an edge between them with a weight of two since two out of three connections share the same register.

The next step of the algorithm selects two nodes at a time and combines them into a super node. This process is applied iteratively until the number of nodes within the graph is equal to the number of functional units available. The partitioning of nodes produces the functional unit binding; all nodes within one super node are bound to the same functional unit.

The process of choosing which two nodes must be combined involves various steps. First, the node with the minimum number of edges is selected. This decision is made so that nodes with limited flexibility are combined first. If nodes with limited flexibility were left to the end then there is a possibility that they may not be able to be combined with other remaining nodes.

The second node, j , in the selection process is selected by giving each node a score in reference to the previously selected node i . The node with the highest score is selected to be combined with i . The score function between i and j is:

$$\text{Score} = \text{weight_score} \times \text{weight}(i,j) + \text{node_score} \times \text{num_nodes}(j) + \text{edge_score} \times \text{num_interference}(j)$$

where $\text{num_nodes}(\text{int})$ is the number of nodes already within node j and $\text{num_interference}(\text{int})$ is the number of other nodes node j interferes with. Furthermore weight_score is many orders of magnitude larger than nodes_score and node_score is many orders of magnitude larger than edge_score . The purpose of this score function is to order nodes in importance of weight first and then by the number of nodes within them and number of interference edges.

After the two nodes, i and j , have been combined into node k , a new set of edges is created between node k and all other nodes in the graph. Edges between node k and another node l exist if and only if both i and j had edges to l before the merge. The weight of the edge between k and l was chosen to be the weighted average below:

$$\text{weight}(k,l) = \frac{\text{weight}(i,l) \times \text{num_nodes}(i) + \text{weight}(j,l) \times \text{num_nodes}(j)}{\text{num_nodes}(i) + \text{num_nodes}(j)}$$

After all nodes have been partitioned, functional unit binding is complete.

2.5 VHDL Exporting

This portion of the code involved using the scheduling, register allocation and functional unit binding to export the datapath and control to VHDL. This portion of the code is not complete. In its place, the code outputs the following information:

1. Functional Unit IDs
2. Scheduling Information: For each BB there is an enumeration of the DAG IDs scheduled at each cycle as well as the functional unit on which they are scheduled on.
3. Register Allocation Information: Indicates which variables and temporary variables are assigned to which register.
4. Bus Driver Locations: Indicates the bus drivers that are needed from each register and from which each functional unit output
5. Total number of Bus Drivers
6. Signals originating from the Control unit going to the Datapath

7. Signals originating from the Datapath going to the Control

2.6 Summary of Functionality

The objective of the project was to perform a behavioral synthesis of a C program and export the results to structural VHDL. While the designed code falls short in exporting to VHDL, this last step is trivial but time-consuming. The designed code first converts the intermediate SimpMeta format to a new intermediate format and then performs the scheduling, register allocation and functional unit binding. The next section will outline some created testbenches and the results obtained from the synthesis tool.

3 Results

To test the created synthesis tool, a set of C programs were created as benchmarks and were used as inputs into the synthesis tool. This set of C programs is found in Appendix 1.

The set of tests included running each C benchmark with various amounts of resources. The complete output from the set of tests, which is too large to be included here, can be seen in the accompanying output.out. The rest of this section will summarize the results from the set of tests.

Table 1 shows the results from the benchmarks. From it certain observations can be made:

1. Program parallelism is restricted heavily by true dependencies within the code, and large increases in functional resources have minimal impact on performance (measured in cycles to completion).
2. Aggressive scheduling, due to the larger availability of resources, may lead to a larger number of registers. This is due to the fact that more operations are occurring simultaneously and therefore more values are needed in tandem increasing the number of edges in the value interference graph.
3. With a larger availability of resources the interconnection networks become larger, not only because of the inherent addition of resources, but also because of the increase in registers noted in the point above.
4. If programs are partitioned well, then additional resources may lead to smaller interconnection networks as both registers and functional units may be clustered with few paths between clusters.

Benchmark	Adders	Multipliers	Dividers	Subtractors	# of Instructions	Cycles	Average Parallelism	# of Variables	# of Registers	Average Number of variables per Register	# of Bus Drivers
1	1	1	0	0	13	7	1.86	5	5	1.00	23
	2	1	0	0	13	7	1.86	5	5	1.00	23
	1	2	0	0	13	6	2.17	5	5	1.00	25
	2	2	0	0	13	6	2.17	5	5	1.00	25
2	1	1	0	0	17	14	1.21	22	10	2.20	28
	2	1	0	0	17	9	1.89	22	11	2.00	36
	1	2	0	0	17	14	1.21	22	10	2.20	28
	2	2	0	0	17	9	1.89	22	11	2.00	36
	3	2	0	0	17	9	1.89	22	12	1.83	35
3	1	1	0	1	14	8	1.75	17	12	1.42	29
	2	1	0	1	14	6	2.33	17	13	1.31	31
	1	2	0	1	14	8	1.75	17	12	1.42	29
	1	1	0	3	14	8	1.75	17	12	1.42	29
	2	2	0	2	14	6	2.33	17	13	1.31	32
4	1	0	1	1	11	7	1.57	5	5	1.00	17
5	1	1	1	1	48	34	1.41	40	19	2.11	61
	2	1	1	1	48	24	2.00	40	19	2.11	67
	1	2	1	1	48	28	1.71	40	19	2.11	61
	2	2	1	1	48	22	2.18	40	19	2.11	67
	3	2	1	1	48	22	2.18	40	19	2.11	72
6	1	1	1	1	13	10	1.30	9	8	1.13	21

Table 1: Summary of Results

4 Conclusion

A synthesis tool that performed scheduling, register allocation, and functional unit binding was created. The tool was tested over a set of benchmarks to observe both its functionality and the characteristics of the benchmarks.

Further optimization may be done on the various heuristics, including constant definitions to improve upon the results.

A1 Appendix 1: Testbenches

Test.c

```
void
behmain(void) {

    int m,n,o,p;

    p = p + p;
    m = n + 2;
    m = n * 3;
    p = 7 * n;

    if ((m + n) == 0) {
        m = 2;
        o = p *p;
        m = n +n;
        n = 6*m;
    }
    else {
        n = 2;
        p = p+p;
        m = o*n;
        n = n*o;
    }
}
```

Test2.c

```
void
behmain(void) {

    int a,b,c,d,e,f,g,h,i,j,k,l,m,n,o;
    int w,x,y,z;

    w = x * y;
    z = x * w;

    o = ((m+n) + (a + b) + (6 + f)) + 3;

    x = o + z;
    e = m + n;
    y = h + (a+b);

    if ( m > 5) {
        o = ((m+a) + (a + m) + (6 + z)) + 3;
    }
}
```

Test3.c

```
void
behmain(void) {

    int a,b,c,d,e,f,g,h,i,j,k,l,m,n,o;
    int w,x,y,z;

    d = l * (j + k);
    a = b + c;
    o = o + o;
    z = z + z;
    y = d - x;
    i = i * d;
    m = a * (j+k);
    h = h + h;
    h = h + h;

    if ((m-n) == 0) {
        a = b + c;
        b = a + c;
    }
}
```

Test4.c

```
int
behMain(int a, int b) {
    int n;
    int i = 0;
    int while_counter = 10;
    int f = a + b;

    switch(i / 5) {
    case 1:
        n++;
        break;
    case 0:
        n--;
        break;
    default:
        n = n + 2;
        break;
    }

    while (while_counter > 0) {
        if (f == 0)
            while_counter--;
        if (while_counter == 1) {
            f = 2;
        }
        if (f == 2) {
            while_counter--;
        }
    }
    return(0);
}
```

Test5.c

```
void
behmain(void) {

    int a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t;
    int w,x,y,z;

    w = x * y;
    z = x * w;

    o = ((m+n) + (a + b) + (6 + f)) + 2;

    x = o + z;
    if (z > 0)
        o = ((m+n) + (a + b)) + 3;
    else
        o = ((m+n) + (5 + f)) + 4;

    e = m + n;
    y = h + (a+b);

    while ((x + y) > (a + b)) {
        p = i+j;
        q = l*m;
        r = o/h;
        s = n+a;
        if (a > b) {
            p += i+j;
            q += l*m;
            r += o/h;
            s += n+a;
        }
        else {
            p *= i+j;
            q *= l*m;
            r *= o/h;
            s *= n+a;
        }
    }
}
```

Test6.c

```
void
behmain(void) {

    int a,b,c,d,e,f,g,h,i,j,k,l,m,n,o;
    int v,w,x,y,z;

    a = a*a;

    if (a > b) {

        if (z > y) {
            c = c*c;
        }
        else {
            d=d*d;
        }
        for (i=0; i < 2; i++)    {
            v=v*v;
            if (h > 9) {
                y=y*y;
            }
            else {
                z=z*z;
            }
        }
    }
    else {
        b=b*b;
    }
}
```