Software Estimation from Executable Specifications

Jie Gong
Daniel D. Gajski
Sanjiv Narayan

Technical Report ICS-93-5
March 8, 1993

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717-3425
(714) 856-8059

Abstract

Previous work in software/hardware codesign has addressed issues in system modeling, partitioning, and mixed module simulation and integration. Software estimation, which provides software metrics to assist software/hardware partitioning, has not been extensively studied. In order to rapidly explore large design space encountered in software/hardware systems, automatic software estimation is indispensable in software/hardware partitioning in which designers or partitioning tools must trade off a hardware with a software implementation for the whole or a part of the system under design. In this report we present a software estimator that provides three software metrics — execution time, program-memory size and data-memory size for a specification executing on a given processor. Experiments have shown that our estimator is fairly accurate when applied to different designs spanning from straight line code to code with branches and loops and even to hierarchical specifications. Experiments also show that our estimator is fast and thus can provide rapid feedback to the designers or system partitioning tools to quickly evaluate different design alternatives.

Contents

List of Figures

1 Introduction

System design is a set of tasks which convert the system-level specification into a set of completely specified interconnected modules implementing the specification. Each module could be implemented in hardware or as software executing on a processor. One of the system design tasks is partitioning of specification into software and hardware parts. A hardware implementation has better performance whereas a software implementation has lower cost, shorter development time and allows changes late in the design cycle. Thus, the most efficient implementation has a minimal amount of costly application-specific hardware while still meeting the required performance constraints.

Several researchers have described frameworks for modeling, simulation and integration of software/hardware designs [SB91][KL92][BV92]. Software/hardware partitioning techniques have been addressed in [GM92a][EH92]. Gupta and De Micheli [GM92a] propose a partitioning algorithm that starts with an initial partition where all operations, except for the unbounded delay operations, are assigned to hardware. The partition is refined by migrating operations from hardware to software in the search for a lower cost feasible partition. The approach used by Ernst and Henkel [EH92] starts with a complete software implementation from which those portions that violate the timing constraints are extracted for hardware implementation. These two approaches start from different directions but work towards the same goal of minimizing the amount of application-specific hardware required. Software/hardware partitioning requires a software estimator that will predict the execution time of the software implementation in order to identify which portion in the specification can be migrated from hardware to software while not violating the constraints or which portion needs to be implemented in hardware to satisfy the timing constraints. To our knowledge, no previous work has addressed the issue of software estimation assisting in the software/hardware partitioning. In the absence of an automatic estimator, the effect of each partitioning can be evaluated only through actual implementation which prevents a designer from considering other design alternatives.

In this report we present our software estimator which provides three software metrics — execution time, program-memory size and data-memory size for a given specification and a given target processor. The generic model used in our estimator does not require different estimators for different target processors. Instead a single estimator and a set of technology files for different target processors are used. This makes our estimator fast and easy to extend for different target processors. Also, the probability-based flow analysis technique used for the performance estimation in our estimator provides further advantages over dynamic simulation [PK90].

The input to our software estimator is a system-level description specified with the executable specification language — SpecChart. A SpecChart consists of hierarchical concurrent/sequential behaviors with leaf behaviors specified using the VHDL language. Details on the language and its constructs may be found in [NVG91].

In the next section, we present the underlying model used for software estimation. Performance and memory size estimation for system-level specifications are discussed in Section 3 and Section 4 respectively. The results of our experiments are presented in Section 5 followed by the conclusion and future work in Section 6.

2 Model for Estimation

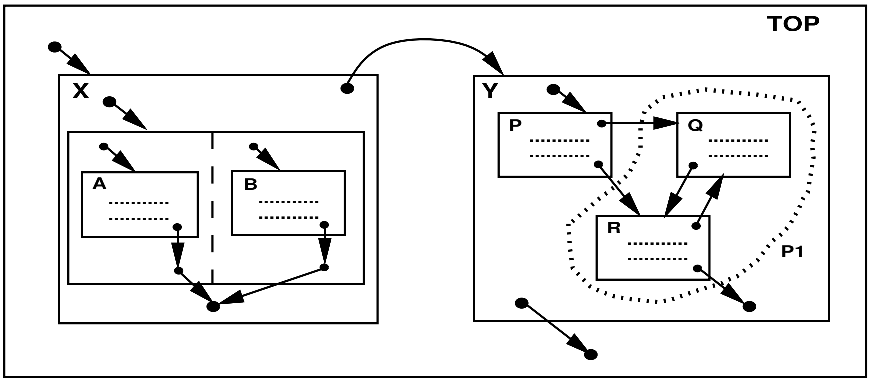


Figure 1: A sample system-level specification

The estimation model we propose is targeted to the system-level specifications which consist of hierarchical concurrent/sequential behaviors. A behavior, which is a set of actions and a set of conditions describing when each action is to occur, can in turn contain sequential or concurrent sub-behaviors. For example, in Figure 1, behavior TOP consists of two sequential sub-behaviors X and Y. Behavior X in turn contains two concurrent sub-behaviors A and B. Behavior Y contains three sequential sub-behaviors P, Q, R. Concurrency is represented by the dashed line like the one between behavior A and B whereas sequencing is represented by those transition arcs. The behavior which does not contain any sub-behaviors is called leaf behavior. In SpecChart, each leaf behavior consists of a set of VHDL sequential statements. Behavior A, B, P, Q and R in Figure 1 are leaf behaviors. The dots in the specification indicate the starting or completion of the behaviors. Our estimators are intended to estimate the software metrics for any given leaf/non-leaf behavior of the specification as well as any given partition (a set of behaviors) in the specification. P1 in Figure 1 is a partition which contains two behaviors Q and R.

2.1 Estimation Model for Leaf Behaviors

In order to obtain the estimates for leaf behaviors, we need to compile the code in the leaf behaviors into machine instructions of the target processor. For example, if a leaf behavior will be implemented on an Intel 8086 processor, it needs to be compiled into the 8086 instruction set. Using the timing and size information associated with each type of instruction such as how many clock cycles the 8086 instruction executes and how many bytes it takes, we can obtain the performance and program size of the behavior. Similarly, if the leaf behavior is going to be implemented on a Motorola 68000 processor, it needs to be compiled into 68000 machine instructions. Based on the 68000 instruction timing and size information, the estimator can obtain the software metrics for the behavior. This model, in which the estimator is targeted to one specific processor, is called processor-specific model (shown in Figure 2(a)).

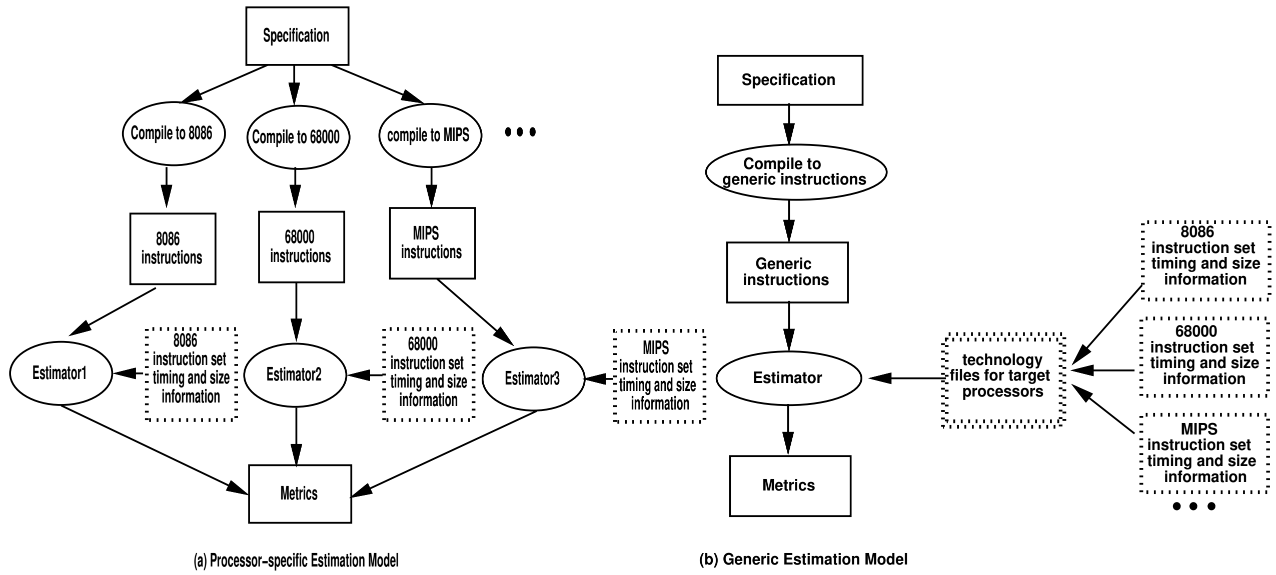


Figure 2: Two different estimation models

Instead of using different compilers and estimators for different target processors in the processor-specific model (Figure 2(a)), we propose a generic estimation model (Figure 2(b)) in which the leaf behavior specification is converted into a set of generic three-address instructions. After that the estimator will compute the software metrics for the leaf behavior based on the generic instructions and the technology files for the target processors. For example, if the leaf behavior is going to be implemented on an Intel 80286 processor, then the technology file for the 80286 processor will be used during the estimation. The technology file for a target processor supplies information about how many clock cycles each type of generic instruction needs and how many bytes it takes if the generic instruction is executed on that target processor. The technology file for a target processor is derived from the timing and size information of the processor's instruction set.

The generic three-address instructions used in our estimation model have the following formats:

1. Arithmetic/logic/relational operation: $\langle$des $\leftarrow$ src1 **op** src2$\rangle$; For unary operations, src1 is empty;

2. Move/load/store operation: $\langle$des $\leftarrow$ src$\rangle$;

3. Conditional jump operation: $\langle$**if** cond **goto** label$\rangle$;

4. Unconditional jump operation: ⟨**goto** label⟩;

5. Procedure call operation: ⟨**call** label⟩;

Here *des*, *src* and *cond* are either constants, registers or memory locations. Memory locations could be directly addressed like A, B or addressed with offset like $A[I]$ or $B[J]$. *label* refers to procedure names or instruction labels. The three-address instructions also include $RETURN$ and $NULL$ instructions.

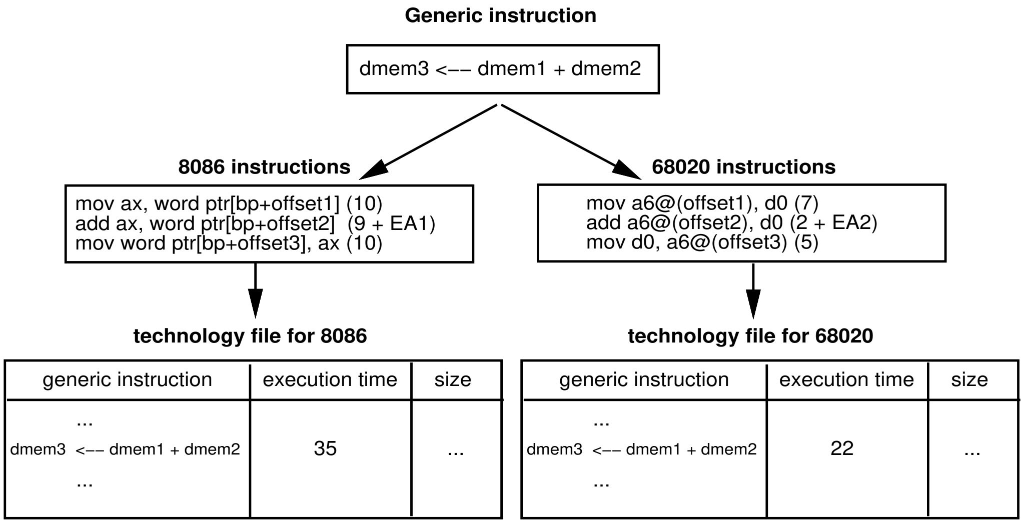


Figure 3: Execution time of the generic instruction for different processors

To provide the technology file for a given processor, we need to find out how many clock cycles each type of generic instruction needs and how many bytes it takes on that processor. Figure 3 shows the computation for the number of clock cycles for the generic instruction with type of $dmem3 \leftarrow dmem1 + dmem2$. Here, $dmem$ indicates direct memory addressing mode. The generic instruction is first mapped to a sequence of target processor instructions followed which the total number of clock cycles of the generic instruction is obtained by summing the clock cycles of each individual instruction in the sequence. $EA1$ and $EA2$ in Figure 3 are the effective address calculation times used for displacement memory addressing mode, which are 6 and 8 clock cycles on the 8086 and 68020 respectively. The generic instruction thus will take 35 and 22 clock cycles on the 8086 and 68020 processors respectively. Using a similar approach we can derive how many bytes each type of generic instruction will take if it is executed on the 8086 or 68000 processor. Presently the technology files for 8086, 80286, 68000 and 68020 processors are supported in our estimator. The 8086, 80286, 68000 and 68020 technology files are derived from the timing and size information of the their corresponding instruction sets given in [I8086][I80286][M68000][M68020]. The technology file for Intel 8086 processor is shown in the Appendix.

Compared with the processor-specific model, our generic model has the following advantages:

1. In the generic model, we do not need to use different compilers and different estimators for

different target processors. Instead, only a single compiler, estimator and a set of technology files are required for the estimation.

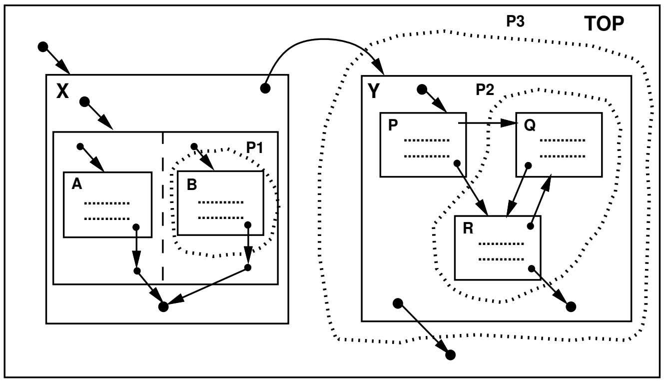
2. The generic model makes it much easier to apply the estimator to other target processors. The estimation can be carried out as long as the technology file for the target processor is supplied.

3. The peculiarities of each type of processor is reflected in the technology file for the processor. The generic three-address instructions are free of instruction idiosyncrasies. Thus it is much easier and faster to compile the specification into the generic instruction set than those associated with specific processors.

2.2 Estimation Model for Non-leaf Behaviors and Partitions

Non-leaf behaviors possess hierarchical or concurrent constructs. To evaluate the software implementation of a given non-leaf behavior or a partition on a specific microprocessor, we must first flatten the hierarchy and sequentialize/schedule the specification to diminish the concurrency [GM92b] since our target machine is a uni-processor. In other words, the specification needs to be mapped (flattened/ sequentialized) into a program written in a language which can be directly compiled to the machine instructions of the given processor. Based on the machine instructions generated, the software metrics such as performance and memory size for the specification can thus be computed. The software metrics obtained in such a way are accurate since they are computed from the actual implementation of the specification on the given processor. However, due to the fact that automatic partitioning tools will evaluate hundreds or thousands of partitions, this approach is too costly and time consuming since we would have to actually implement each partition on the given processor through flattening, sequentializing and compiling process in order to get the software estimates for that partition. To get fast estimates while not sacrificing too much accuracy, the estimation model we propose combines two different approaches: an accurate approach for estimating leaf behaviors and a fast approach for estimating non-leaf behaviors and partitions. Prior to the partitioning process, each leaf behavior is compiled and estimated using the approach described in the previous section. During the partitioning process, the software estimates for each partition are constructively computed bottom up from the estimates of the leaf behaviors. Such a combined approach is very fast because it does not involve flattening, scheduling, compiling for each partition during the design process. It only requires some computation based on the pre-obtained estimates for the leaf behavior specifications. Therefore this model allows rapid evaluation of different design alternatives.

Figure 4 illustrates our estimation model with a sample SpecChart. Before partitioning process, the software metrics for leaf behaviors A, B, P, Q, and R are estimated. During partitioning process, the software metrics for each partition are computed based on the already known estimates of its constituent behaviors. For example, the estimates of partition P2 is a function of the estimates of behavior Q and R.



Partitions	Estimates
A, B, P, Q or R	known
X	func(A, B)
Y	func(P, Q, R)
P1	func(B)
P2	func(Q, R)
P3	func(Y)

Figure 4: A sample specChart and its estimates

3 Performance Estimation

There are two possible ways to find out how many clock cycles a program takes to execute — dynamic simulation and static estimation. Given a set of input data, dynamic simulation actually executes the program and records the clock cycles used in each execution. Given different sets of input data, dynamic simulation may obtain different number of clock cycles for that program due to the data dependent conditional branches and loops. Static estimation, on the other hand, is insensitive to input data. It just computes the average number of clock cycles needed to execute the program. Static estimation can yield good results if the number of loop iterations is known and the conditional branching probability can be predicted correctly. Besides, static estimation has a number of advantages: (1) It takes much less time and space than dynamic simulation. (2) It does not need input data (test vectors). The probability-based flow analysis technique and its application to the performance estimation for system-level specifications are presented in this section.

3.1 Flow Analysis

Flow analysis is a technique used in the static estimation for design with conditional branching (including loops). Given a control flow graph $G = (V, E)$ representing a portion of the design, where V is the set of vertices v_i, and E is the set of *directed* edges e_{ij} connecting vertex v_i to v_j and indicating sequencing between v_i and v_j, we wish to determine the execution frequencies of each of its nodes based on the branching probabilities. By determining the execution frequencies of the nodes, we can obtain useful information about the design by associating with each node in the graph, *weight* representing some design parameters. For example, if each node weight represents the execution time of a basic block derived from the code of a leaf behavior, then we can use the execution frequency of the nodes and the weights of the nodes to determine the total execution time for the leaf behavior by taking a weighted sum of these two quantities.

3.1.1 Determining the Branching Probabilities

Branching probabilities are associated with the edges in the control flow graph. They could be determined in the following ways:

1. **Equal Probabilities** : In case of branching, we assign equal probabilities to all the edges emerging from the node. Thus, if there are n edges emerging from a node, all of them are assigned a probability of $1/n$.

2. **Loop Related Probabilities** : When the number of loop iterations is known, say n, the exit edge has a probability of $1/n$ while the back edge has a probability of $(n-1)/n$.

3. **User Defined Probabilities** : The user may specify the branch probabilities in the SpecChart description using annotations.

4. **Simulation Based Probabilities** : For any given set of input data, a record can be kept of the branches taken during the simulation. From this observed behavior, the probabilities of the branches can be derived.

Currently the first three approaches have been implemented in our estimator.

3.1.2 Determining Node Execution Frequencies

The execution frequency of a node is defined as the number of times on the average that the node will be executed in a single execution of the graph. We use the branching probabilities between the nodes to determine the execution frequencies. This is given in the following procedure.

Determining Node Execution Frequencies:

1. Determine the branch probabilities using one of the methods outlined above.

2. A start node, S, preceding the first node in the graph, is added. Its execution frequency, $F(S)$ is set to 1 since this node is executed exactly once whenever the control flow graph is executed.

3. The execution frequency $F(N_j)$ for any node N_j depends on the weighted execution frequencies of all its immediate predecessor nodes. The execution frequency for each predecessor node N_i is multiplied by the branch probability of the edge between N_i and N_j, $P(e_{ij})$. For each node in the graph, we first formulate the equation for the node execution frequency.

$$F(N_j) = \sum_{all\ predecessor\ nodes\ N_i\ of\ N_j} F(N_i) \times P(e_{ij}) \qquad (1)$$

4. We then solve the set of equations formulated in step 3 to obtain the individual execution frequencies. There are a variety of methods such as Gaussian Elimination, LU decomposition, Chomsky's method which can be used. We have selected the Gaussian Elimination method in our estimator.

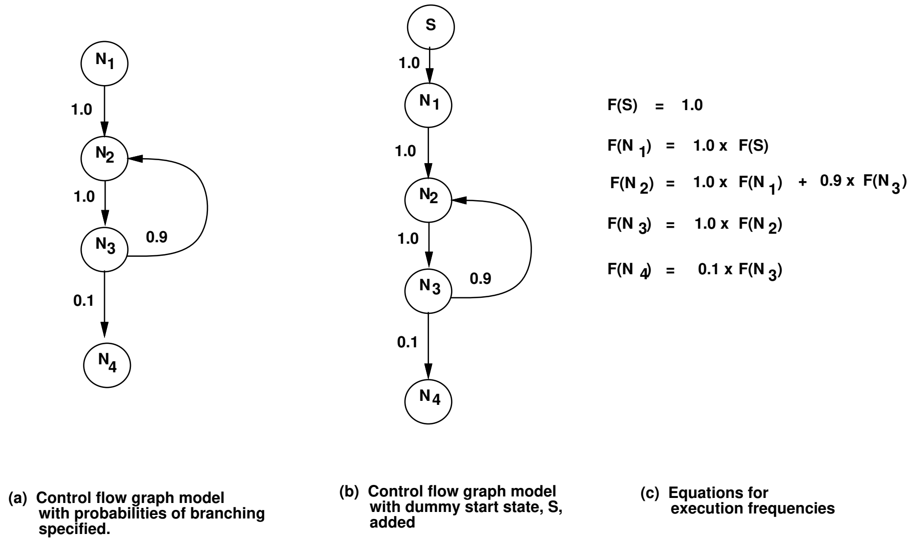


Figure 5: Obtaining a set of linear equations from the control flow graph model

The procedure is illustrated by an example shown in Figure 5. A control flow graph with the branch probabilities is shown in Figure 5(a). Figure 5(b) shows the same graph with the added dummy node, S. Figure 5(c) shows the equations for the execution frequency of each node. As an example, consider node N_2. It has two predecessor nodes N_1 and N_3. The probabilities of the edges e_{12} and e_{32} are 1.0 and 0.9 respectively. Thus the equation for the execution frequency for N_2 is :

$$F(N_2) = 1.0 \times F(N_1) \; + \; 0.9 \times F(N_3) \tag{2}$$

Solving the equations in Figure 5(c) yields the following values for execution frequency :

$$F(N_1) = 1, \quad F(N_2) = 10, \quad F(N_3) = 10, \quad F(N_4) = 1$$

These are the expected execution frequencies returned after probability based analysis of the control flow graph.

In situations where the edges are also associated with some weights (as the case when conditions exist on arcs between nodes), we may need to know the execution frequency of each edge $F(e_{ij})$. It is obvious that the execution frequency of an edge is the same as that of its target node. Therefore we have $F(e_{ij}) = F(N_j)$.

3.1.3 Determine the Performance of Control Flow Graph

For a control flow graph $G = (V, E)$ with each node $v \in V$ associated with a weight $W(v)$ and each edge $e \in E$ associated with a weight $W(e)$, if we know the execution frequency of each node $F(v)$ and the execution frequency of each edge $F(e)$ in the graph, then the performance $P(G)$ of the graph G

can be calculated as follows.

$$P(G) = \sum_{for\ all\ v \in V} (W(v) \times F(v)) + \sum_{for\ all\ e \in E} (W(e) \times F(e)) \tag{3}$$

3.2 Performance Estimation for Leaf Behavior

3.2.1 Constructing Basic Blocks from VHDL statements

We divide the VHDL code segment in each leaf behavior into *basic blocks*. A *basic block* [ASU88] is a sequence of consecutive VHDL statements in which flow of control enters at the beginning and leaves at the end without halting or the possibility of branching, except at the end. To determine the basic blocks, we first determine the set of *leaders*, the first statement of a basic block. The rules we use to determine leader statements are: (1) The first statement of the code segment is a leader. (2) All wait statements and procedure calls are leaders. (3) Any statement which is the target of a conditional statement (if, loop, case) is a leader. The target of a conditional statement is any statement to which control could possibly be transferred on evaluating the condition. (4) Any statement that immediately follows a conditional statement, wait statement, or a procedure call is a leader.

For each leader determined above, its basic block consists of the leader and all statements up to but not including the next leader (or the end of the given VHDL code segment). A basic block will then contain one of the following: (1) a set of assignment statements, or (2) a single wait statement, or (3) a single procedure call.

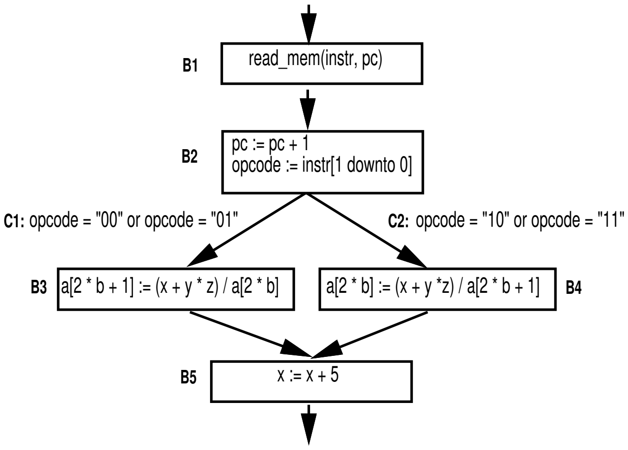
Figure 6 shows how the basic blocks can be constructed from a given set of sequential VHDL statements. In Figure 6(a), the leaders are denoted by horizontal arrows along with the corresponding rule number used to determine the leader. Figure 6(b) shows the basic block structure of the VHDL code. The conditions to be evaluated for the conditional branching are associated with the edges between basic blocks.

3.2.2 Obtaining Weights for Basic Blocks and Conditions

Once basic blocks have been determined (Figure 6(b)), we need to find the weight (i.e. execution time) of each basic block and weight of each condition. The VHDL assignment statements in each basic block and condition are compiled into our generic three-address instructions. Figure 7(a) shows the basic block structure of Figure 6(b) after the VHDL code is compiled into generic instructions. The execution time of each generic instruction can be obtained from the technology file based on its type. For example, the instruction $pc \leftarrow pc + 1$ has the type of $dmem \leftarrow dmem + constant$ ($dmem$ means directly memory addressing mode). In Figure 7(a), $T1$, $T2$ and $T3$ are temporary variables which can be considered as type of *register* or *dmem* depending on how many general-purpose registers the target processor has. Suppose after the compilation, m temporary variables $T1, T2, \ldots, Tm$ are

```
1,2 –>   read_mem(instr, pc);
  4 –>   pc := pc + 1;
         opcode := instr[1 downto 0];
         if (opcode = "00" or opcode = "01") then
  3 –>      a[2 * b + 1] := (x + y * z) / a[2 * b];
         elsif (opcode = "10" or opcode = "11") then
  3 –>      a[2 * b] := (x + y * z) / a[2 * b + 1];
         endif;
  4 –>   x := x + 5;
```

(a)



(b)

Figure 6: Building basic blocks from VHDL statements

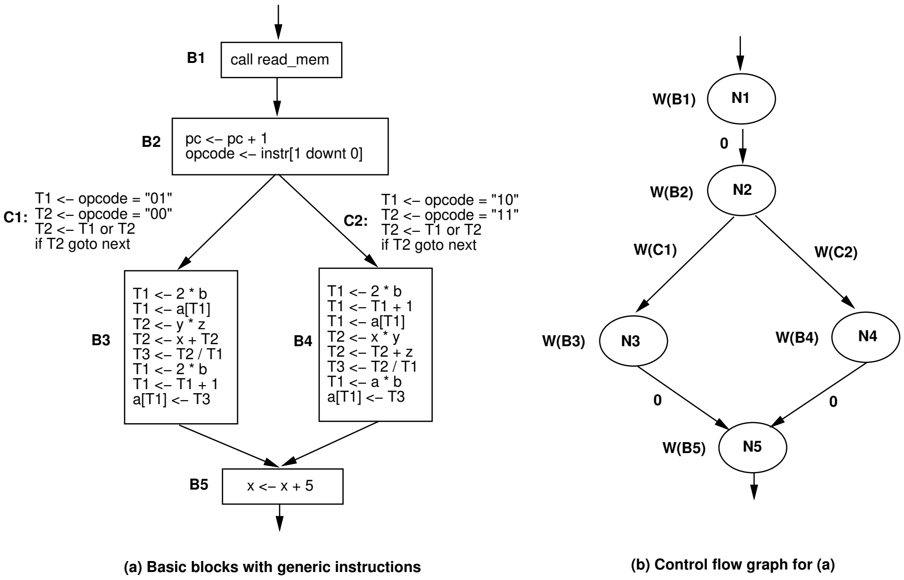


Figure 7: A basic block structure and its corresponding control flow graph

used and the target processor has n general-purpose registers. If $n \geq m$, then $T1, T2, \ldots, Tm$ will be considered as register type, otherwise, $T1, T2, \ldots, Tn$ will be considered as register type whereas $Tn+1, \ldots, Tm$ will be considered as *dmem* type. Temporary variable with smaller index (e.g. $T1$) will be used more heavily than temporary variable with larger index (e.g. $T9$) in the compilation. Therefore temporary variable with smaller index should have higher priority to be assigned with a register. In the example shown in Figure 7(a), suppose the target processor has two general-purpose registers. So $T1$ and $T2$ will be considered as register type whereas $T3$ will be considered as a memory type. Thus instruction $T1 \leftarrow T1 + 1$ has the type of $register \leftarrow register + constant$. Instruction $T3 \leftarrow T1/T2$ has the type of $dmem \leftarrow register/register$. The weight $W(B)$ of a basic block B can be computed by summing the execution time of each generic instruction in that basic block. The weight $W(C)$ of a condition C can be obtained in a similar way.

3.2.3 Obtaining Weight of Basic Block With a Wait Statement

The weight of a basic block with a wait statement is computed as follows:

1. **Wait :** In the absence of a timeout clause, the execution time is set to a very large number like MAXINT, the largest integer supported by the host machine.

2. **Wait on** $S_1, ...S_n$ **until** $C_1, ...C_2$ **for T ns :** The execution time for a wait statement with a *timeout clause* (i.e. *for* clause) is equivalent to the smallest multiple of the clock period higher than T. The effect of conditions (e.g. C1) and an event on signals in the sensitivity list (e.g. S1) are ignored.

3. **Wait on** $S_1, ...S_n$ **until** $C_1, ...C_2$ **:** i.e wait statement without a timeout clause. If the wait statement has a sensitivity list or condition clause but no timeout clause then the wait statement requires at least one clock cycle to be executed. The effect of conditions (e.g. C1) and an event on signals in the sensitivity list (e.g. S1) are ignored.

3.2.4 Obtaining Weight of Basic Block with a Procedure call

The third type of basic block is one which has a single procedure call. The execution time for the procedure can be determined by treating the body of the procedure declaration as a leaf state. This time is then used as the execution time of the basic block which contains the procedure call.

3.2.5 Computing Performance for Leaf Behavior

The basic block structure of a leaf behavior is mapped to an equivalent control flow graph G. Figure 7(b) shows the corresponding control flow graph of Figure 7(a). Each basic block B_i is mapped to a node N_i in G. Each edge connecting two basic blocks B_i and B_j is mapped to an edge connecting node N_i and N_j in the graph. Each node N_i in G has a weight which is the same as $W(B_i)$. Each edge in G has a weight which is the same as the weight of the condition associated with the corresponding edge in the basic block structure (see $c1$ and $c2$ in Figure 7). Applying the flow analysis and using the weights obtained for the nodes and edges, the average execution time for the leaf behavior can be computed using equation 3 in section 3.1.3.

3.3 Performance Estimation for Non-leaf Behavior

Once execution times have been estimated for each of the leaf behaviors as shown above, we can merge the performance estimates of the leaf behaviors to yield the the performance estimate of the next higher behavior in the hierarchy. The approach adopted is similar to that of merging the performance estimates of basic blocks to obtain performance estimates for the leaf behaviors. At any level of the hierarchy, we will first determine the performance estimates of the child behaviors and then combine these estimates to determine the performance estimate of the parent behavior.

To estimate the performance for B_{parent}, a non-leaf behavior with sequential sub-behaviors, we create a control flow graph $G = (V, E)$ for its child behaviors whose performance estimates are already known. For each of the sub-behaviors, B_i, of B_{parent}, there exists a corresponding vertex v_i in the graph G. For every transition arc between the two sub-behaviors B_i and B_j, the set E has a directed edge e_{ij} from vertex v_i to vertex v_j in G. After the control flow graph model has been constructed for the sub-behaviors, we can apply the flow analysis (section 3.1.3) to obtain the performance of the parent behavior B_{parent}.

In case a behavior at any level of the hierarchy has concurrent sub-behaviors, the execution time of that behavior is computed as the sum of that of its child behaviors since the concurrent child behaviors have to be sequentialized to execute on a uniprocessor [GM92b]. It must be mentioned here that a non-leaf behavior may have a descendant sub-behavior which does not have a *stop dot* in SpecChart. In this case the behavior will never finish executing and consequently the execution time returned for that behavior is an arbitrarily large number.

3.4 Summary of Performance Estimation

Figure 8 summarizes the performance estimation for different entities such as *partition*, *behavior* of the specification, *basic block* of the leaf behavior and *generic instruction* of the basic block. The performance of a partition depends on that of its constituent behaviors. The performance of a sequential non-leaf behavior is determined using flow analysis on the performance of its constituent sub-behaviors. The performance of a concurrent non-leaf behavior is the sum of the performance of its constituent sub-behaviors. The performance of a leaf behavior is determined using flow analysis on the performance of its constituent basic blocks. The performance of each basic block is computed by summing the performance of its constituent generic instructions. The performance of each generic instruction is taken from the technology files supplied to the estimator.

Entity to be estimated	Components of entity	Weights of components	Technique to be applied
partition	behaviors	execution times of behaviors	sum/flow analysis
sequential non-leaf behavior	sub-behaviors	execution times of sub-behaviors	flow analysis
concurrent non-leaf behavior	sub-behaviors	execution times of sub-behaviors	sum
leaf behavior	basic blocks	execution times of basic blocks	flow analysis
basic block	generic instructions	execution times of generic instructions	sum
generic instruction	——	execution times specified in the technology file	——

Figure 8: Performance estimation for different entities

4 Memory Size Estimation

Given a behavior, memory size estimation is to determine how much program-memory (i.e. bytes used to store the compiled program representing the behavior) and how much data-memory (i.e. bytes used to store the data manipulated by the behavior) are needed.

4.1 Program-memory Size Estimation

The size of each type of generic instruction is specified in the technology file for target processor. For example, the size of the generic instruction with type of $dmem1 \leftarrow dmem2 + dmem3$ is 9 bytes in the technology file for the 8086 processor and 6 bytes in the technology file for the 68020 processor. Based on the size of each generic instruction, the program-memory size of each basic block is the sum of that of all generic instructions in that basic block. The program-memory size of a leaf behavior in turn is the sum of that of all its basic blocks as well as the size of all conditions associated with the edges between basic blocks. Analogously, the program-memory size of a non-leaf behavior is the sum of that of all its sub-behaviors as well as the size of all conditions associated with the arcs between its sub-behaviors.

4.2 Data-memory Size Estimation

The data-memory size is determined based on the data declaration parts in the specification. The data-memory size $DMS(D)$ of a declaration D is determined by the size of D's base type and the number of base type elements in D. For example, let's consider the following two declarations:

```
(a) variable a: integer;
(b) variable b: bitvector [9 downto 0];
```

The base type of the declaration (a) is *integer* and the number of base type elements in (a) is 1. The base type of (b) is *bitvector* and the number of base type elements in (b) is 10.

The data-memory size of a declaration D is computed as follows:

$$DMS(D) = DMS(BT(D)) \times N \tag{4}$$

where, $BT(D)$ is the base type of the declaration D. N is the number of base type elements in D.

Base Type	Data Memory Size (bytes)
Bit	1
Bitvector	$\lceil n/8 \rceil$ where n is the number of bits in the vector.
Boolean	1
Character	1
Integer	4
Natural	4
Positive	4
Real	8
String	4
Time	4

Figure 9: Size of the base type

The data-memory size of each base type is specified in a configuration file. The information currently used in the configuration file is shown in Figure 9.

After obtaining the data-memory size of each declaration, The data-memory size of a leaf behavior B can be computed by summing that of each declaration in its declaration part $Decl(B)$. The data-memory size $DMS(B_{parent})$ of a non-leaf behavior B_{parent} is given as follows:

$$DMS(B_{parent}) = \sum_{all\ sub-behaviors\ B_i\ of\ B_{parent}} DMS(B_i) \ + \sum_{all\ declarations\ D_i\ in\ Decl(B_{parent})} DMS(D_i) \tag{5}$$

5 Results

Given a SpecChart description and a leaf behavior name in the SpecChart, our estimator will output the estimates for each basic block in the leaf behavior and the execution frequency of each basic block as well as the total estimates for the leaf behavior. In this way, designers who are not only interested in the estimates of the whole behavior but also interested in the estimates of a loop body or some basic blocks can easily find out estimates of any part of the specification. Similarly, given a SpecChart description and a non-leaf behavior name, our estimator will output the estimates and execution frequency for each sub-behavior in the non-leaf behavior as well as the total estimates of the non-leaf behavior.

We have compared the estimation results for different target processors including 8086, 80286, 68000, 68020 with those results obtained by flattening (this step is not needed in the first two designs) and compiling the designs directly into the instruction set of the target processors. Since there is no VHDL compiler available for those target processors, we first manually converted the VHDL code to its equivalent C code. Following that we compiled the C code into the instruction set of 8086, 80286, 68000 and 68020 processors. Based on the machine instructions generated from the C compilers and the instruction timing and size information provided in [I8086][I80286][M68000][M68020], we have manually computed the actual performance and program-memory size for the designs. Those actual results were then compared with the estimates based on the generic three-address instruction compiler supplemented by corresponding technology files for the target processors.

The first two descriptions we choose were the fifth order elliptical filter and the differential equation example adopted from [DR92]. The behavior of the elliptical filter consists of an infinite loop. We estimated only the loop body. In the differential equation example, there are three basic blocks and one of them is a loop. The loop body can be executed any number of times depending on the external parameters. In our experiments, we have assumed that the loop body would be executed 10 times. The performance results are shown in Figure 10 while program-memory sizes are shown in Figure 11. The data-memory size estimated for the elliptical filter example is 90 bytes. The data-memory size estimated for the differential equation example is 21 bytes.

Application	Target Machine	Actual Performance (in clock cycles)	Estimated Performance (in clock cycles)	Estimation Error
Elliptic filter	8086	2569	2488	-3.2%
Elliptic filter	80286	712	662	-7.0%
Elliptic filter	68000	1692	1632	-3.5%
Elliptic filter	68020	924	888	-3.9%
Differential equation	8086	9446	10586	12.1%
Differential equation	80286	2244	2304	2.7%
Differential equation	68000	6112	6452	5.6%
Differential equation	68020	3416	3676	7.6%

Figure 10: Performance estimation

Generally, compilers optimize the object code by using different optimization techniques such as global optimization, loop optimization, register allocation, optimization for speed or space. Users

Application	Target Machine	Actual Instruction memory size (in bytes)	Estimated Instruction memory size (in bytes)	Estimation Error
Elliptic filter	8086	336	319	-5.1%
Elliptic filter	80286	336	319	-5.1%
Elliptic filter	68000	230	214	-7.0%
Elliptic filter	68020	229	213	-7.0%
Differential equation	8086	151	143	-5.3%
Differential equation	80286	147	141	-4.1%
Differential equation	68000	106	98	-7.5%
Differential equation	68020	105	97	-7.6%

Figure 11: Program-memory size estimation

can invoke those optimizations by passing special flags to the compiler. In previous experiments, we have disabled those optimizations during the C compilation since our generic compiler does not use optimization heuristics. Therefore our estimates are for those non-optimized code. In order to estimate for the optimized code, we need to know the optimization ratio of the compiler to be used by the designer to generate the machine instructions. The performance-optimization ratio α is defined as performance of the optimized code over the performance of the non-optimized code. The size-optimization ratio, β, is defined similarly. To obtain the optimization ratio for each compiler, we have performed several experiments and obtained average optimization ratios for the four compilers used in our experiments (Figure 12).

Compiler	Optimization ratio for performance	Optimization ratio for size
Cto8086	0.74	0.58
Cto80286	0.68	0.56
Cto68000	0.54	0.54
Cto68020	0.49	0.51

Figure 12: Optimization ratio of the compilers

After knowing the performance and size optimization ratio α and β, the estimates for the optimized code can be obtained by multiplying the estimates of the non-optimized code (obtained from our estimator) with α or β. Figure 13 compares our estimates with the actual performance of the optimized code compiled from the designs.

Application	Target Machine	Actual Performance (in clock cycles)	Estimated Performance (in clock cycles)	Estimation Error
Elliptic filter	8086	1984	1841	-7.2%
Elliptic filter	80286	483	450	-6.8%
Elliptic filter	68000	1020	881	-13.6%
Elliptic filter	68020	449	435	-12.9%
Differential equation	8086	9282	7834	-15.6%
Differential equation	80286	1690	1567	-7.3%
Differential equation	68000	3986	3484	-12.6%
Differential equation	68020	1970	1801	-8.6%

Figure 13: Performance estimation of optimized code

The next design we experimented is the real-time medical system used to measure a patient's bladder volume described in [Wu85]. The SpecChart description of this medical system is shown in Figure 14.

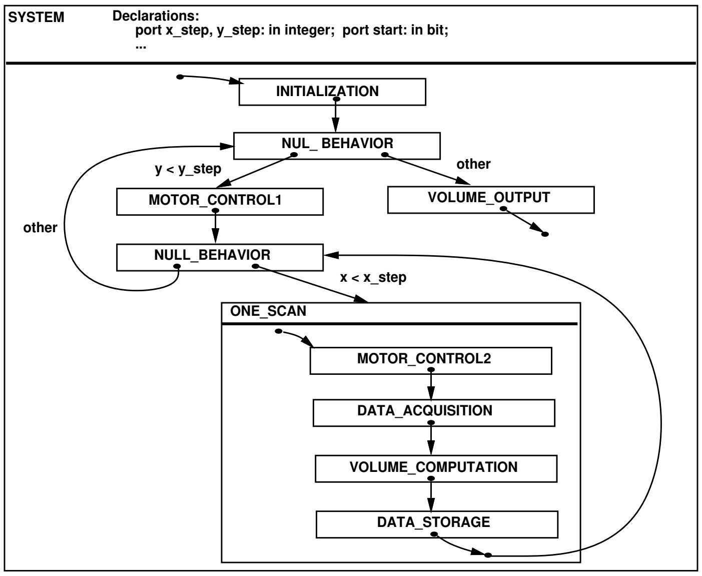


Figure 14: SpecChart of a medical system

There are two timing constraints imposed on this medical system. One is associated with the behavior DATA_ACQUISITION, which requires that acquisition and conversion of 1000 data points take place in less than 1 ms. The other is associated with behavior ONE_SCAN, which requires that the maximum time between two scans, i.e. the time used to execute MOTOR_CONTROL2, DATA_ACQUISITION, VOLUME_COMPUTATION and DATA_STORAGE, is 1 second. We have estimated behavior DATA_ACQUISITION and behavior ONE_SCAN using our estimator. The estimates are compared with the actual results obtained from the (non-optimized) target machine instructions (Figure 15).

Application	Target Machine	Actual Performance (in clock cycles)	Estimated Performance (in clock cycles)	Estimation Error
DATA_ ACQUISITION	8086	82015	71004	-13.4%
DATA_ ACQUISITION	80286	27056	24002	-11.3%
DATA_ ACQUISITION	68000	92089	88012	-4.4%
DATA_ ACQUISITION	68020	44044	41006	-6.9%
ONE_SCAN	8086	189914	153641	-19.1%
ONE_SCAN	80286	65931	55580	-15.7%
ONE_SCAN	68000	198665	175620	-11.6%
ONE_SCAN	68020	98982	84531	-14.6%

Figure 15: Performance estimation

Since ONE_SCAN is a non-leaf behavior, we need to flatten it into a leaf behavior first. Following that the VHDL code is translated into C and compiled into the target machine instructions. For DATA_ACQUISITION, the flattening step is not needed since it is a leaf behavior by itself. Because of no wait statements in C, during the translation from VHDL to C, wait statements were substituted with dummy procedure calls. When we computed the actual performance from the machine instructions, we substituted the performance of those dummy procedure call portions with the performance of the corresponding wait statements obtained from our estimator.

In this experiment, equal branching probability are assumed for all the branches in the behaviors. The number of loop iteration in DATA_ACQUISITION were 1000 which is the number of data needed to be sampled in each scan. The loop iteration numbers in the VOLUME_COMPUTATION were set to 200 for computing the flat level of the bladder, 400 each for computing the anterior wall and the posterior wall of the bladder. The number of loop iteration in DATA_STORAGE were the same as number of data fetched, which is 1000.

If the behavior DATA_ACQUISITION is going to run on an 8086 microprocessor with 12 MHZ clock rate, the estimator predicted 71004 clock cycles which is equivalent to 5.9 ms in this case. Since the timing constraint (1 ms) imposed on DATA_ACQUISITION was violated, the partitioner will conclude that custom hardware must be designed for this behavior. The timing constraint of 1 second imposed on the behavior of ONE_SCAN has not been violated since it only requires 12.8 ms (153641 clock cycles) to execute all behaviors in ONE_SCAN on the chosen microprocessor. If behavior DATA_ACQUISITION is extracted out and implemented by some faster design, the execution time for behavior ONE_SCAN can be expected to be less than 12.8 ms. Therefore all behaviors in ONE_SCAN except DATA_ACQUISITION can be implemented using the code running on the microprocessor.

It took 0.54, 0.33, 0.96 and 1.65 seconds on a Sun4 system to estimate the performance of the elliptic filter, differential equation, DATA_ACQUISITION and ONE_SCAN modules respectively in conjunction with the 8086 technology file. On the contrary, it took several days to manually compute the same information.

6 Conclusion

In this report we have presented techniques for estimating from executable specification the performance and memory size of software code running on a given processor. The experiments have shown that our estimator has an average error of 7.4% and has a maximum error of 19.1% on designs spanning from straight line code (elliptic filter) to code with branches and loops (differential equation and DATA_ACQUISITION module) and even to hierarchical specification (ONE_SCAN module). Experiments also show that our estimator is very fast and can provide rapid feedback to the designers or system partitioning tools and enable them to explore larger design space, which may lead to faster and/or cheaper designs.

Since the generic three-address instructions and the technology files can only characterize the target machine instructions to some extent, there is always a difference between the estimates obtained from our estimator and the results obtained directly by compiling to target machine instructions. We can expect more accurate estimation by enhancing the technology files to include more information about the target machine instruction sets.

Currently our generic instruction set has limited formats, especially in terms of memory addressing modes. If we can incorporate more memory addressing modes in our compiler to close the gap between the generic instructions and the target machine instructions, we can expect better estimation results. However this may increase the complexity of the generic instructions. Increasing complexity of the generic instructions may increase the compiling time and hence increase the whole estimation time. Therefore more studies are needed to investigate what constitutes a suitable generic instruction set.

And, the estimation used for the *wait* statements is very rudimentary. Future work is needed to obtain better estimation for the *wait* statements through considering the input rates of the signals in the sensitivity list.

7 Acknowledgements

This work was supported by the National Science Foundation (grant #MIP-8922851) and the Semiconductor Research Corporation (grant #91-DJ-146). We are grateful for their support. The authors would like to thank Frank Vahid for for his helpful suggestions.

8 References

[ASU88] A. Aho, R. Sethi, and J. Ullman, “Compilers Principles,Techniques, and Tools”, Addison Wesley, 1988.

[BV92] K. Buchenrieder, C. Veith, “ CODES: A practical concurrent design environment”, the International Workshop on Hardware/Software Codesign, Sept. 1992, Estes Park, Colorado, USA.

[DR92] N. Dutt and C. Ramachandran, “Benchmarks for the 1992 high level synthesis workshop”, UC Irvine, Dept. of ICS, Technical Report 92-107, 1992.

[EH92] R. Ernst, J. Henkel, “Hardware-software codesign of embedded controllers based on hardware extraction”, the International Workshop on Hardware/Software Codesign, Sept. 1992, Estes Park, Colorado, USA.

[GM92a] R.K. Gupta, G. De Micheli, “System-level synthesis using re-programmable components”, EDAC’92, Sept. 1992.

[GM92b] R.K. Gupta, G. De Micheli, “Program implementation schemes for hardware-software systems”, the International Workshop on Hardware/Software Codesign, Sept. 1992, Estes Park, Colorado, USA.

[I80286] L. Scanlon, “8086/8088/80286 Assembly Languages”, Simon & Schuster, 1988.

[I8086] G. Gorsline, “16-Bit Modern Microcomputers: the INTEL I8086 Family”, Prentice-Hall, 1985.

[KL92] A. Kalavade, E.A. Lee, “ Hardware/software co-design using ptolemy – a case study”, the International Workshop on Hardware/Software Codesign, Sept. 1992, Estes Park, Colorado, USA.

[M68000] J. Bennett, “68000 Assembly Language Programming: A Structured Approach”, Prentice-Hall, 1987.

[M68020] MOTOROLA, “MC68020: 32-Bit Microprocessor User’s Manual (Second Edition)”, Prentice-Hall, 1987.

[NVG91] S. Narayan, F. Vahid, and D. Gajski, "System specification and synthesis with the SpecCharts Language", ICCAD, Nov. 1991.

[PK90] R. Puschner, C. Koza, "Calculating the maximum execution time of real-time programs", Journal of Real-time Systems, Kluwer Academics, 1989, P159-176.

[SB91] M.B. Srivastava, R. W. Brodersen, "Rapid-prototyping of hardware and software in a unified framework", ICCAD, Nov. 1991.

[W85] A. Wu, "A Microprocessor–based ultrasonic system for measuring bladder volumes", Master Thesis in Electrical and Computer Engineering at University of Arizona, Tucson, 1985.

A Technology Files

```
## Anything after '#' are comments.
## This is the technology file for 8086 processor.
## DirectMem means direct memory addressing.
## IndirectMem means indirect memory addressing.
# OP    DESTINATION    SOURCE1     SOURCE2     time(clock cycles)  size(bytes)
  ALU     Register     Constant     Constant                  4            3
  ALU     Register     Constant     Register                  7            5
  ALU     Register     Register     Constant                  6            6
  ALU     Register     Register     Register                  5            4
  ALU     Register    DirectMem     Constant                 17            7
  ALU     Register     Constant    DirectMem                 18            6
  ALU     Register    DirectMem     Register                 16            5
  ALU     Register     Register    DirectMem                 16            5
  ALU     Register    DirectMem    DirectMem                 27            6
  ALU     Register  IndirectMem     Constant                 20            8
  ALU     Register     Constant  IndirectMem                 21            7
  ALU     Register  IndirectMem     Register                 19            6
  ALU     Register     Register  IndirectMem                 19            6
  ALU     Register  IndirectMem    DirectMem                 30            7
  ALU     Register    DirectMem  IndirectMem                 30            7
  ALU     Register  IndirectMem  IndirectMem                 33            8
  ALU    DirectMem     Constant     Constant                 15            5
  ALU    DirectMem     Constant     Register                 21            8
  ALU    DirectMem     Register     Constant                 20            9
  ALU    DirectMem     Register     Register                 19            7
  ALU    DirectMem    DirectMem     Constant                 31           10
  ALU    DirectMem     Constant    DirectMem                 32            9
  ALU    DirectMem    DirectMem     Register                 30            8
  ALU    DirectMem     Register    DirectMem                 30            8
  ALU    DirectMem    DirectMem    DirectMem                 41            9
  ALU    DirectMem  IndirectMem     Constant                 34           11
  ALU    DirectMem     Constant  IndirectMem                 35           10
  ALU    DirectMem  IndirectMem     Register                 33            9
  ALU    DirectMem     Register  IndirectMem                 33            9
  ALU    DirectMem  IndirectMem    DirectMem                 44           10
  ALU    DirectMem    DirectMem  IndirectMem                 44           10
  ALU    DirectMem  IndirectMem  IndirectMem                 47           11
  ALU     Register        Empty     Constant                  4            3
  ALU     Register        Empty     Register                  5            4
  ALU     Register        Empty    DirectMem                 16            5
  ALU     Register        Empty  IndirectMem                 19            6
  ALU    DirectMem        Empty     Constant                 21            8
  ALU    DirectMem        Empty     Register                 19            7
  ALU    DirectMem        Empty    DirectMem                 30            8
  ALU    DirectMem        Empty  IndirectMem                 33            9
```

MUL	Register	Constant	Constant	4	3
MUL	Register	Constant	Register	74	7
MUL	Register	Register	Constant	72	6
MUL	Register	Register	Register	135	4
MUL	Register	DirectMem	Constant	83	7
MUL	Register	Constant	DirectMem	79	7
MUL	Register	DirectMem	Register	146	5
MUL	Register	Register	DirectMem	146	5
MUL	Register	DirectMem	DirectMem	157	6
MUL	Register	IndirectMem	Constant	86	8
MUL	Register	Constant	IndirectMem	82	7
MUL	Register	IndirectMem	Register	149	6
MUL	Register	Register	IndirectMem	149	6
MUL	Register	IndirectMem	DirectMem	160	7
MUL	Register	DirectMem	IndirectMem	160	7
MUL	Register	IndirectMem	IndirectMem	163	8
MUL	DirectMem	Constant	Constant	15	5
MUL	DirectMem	Constant	Register	88	10
MUL	DirectMem	Register	Constant	86	9
MUL	DirectMem	Register	Register	149	7
MUL	DirectMem	DirectMem	Constant	97	10
MUL	DirectMem	Constant	DirectMem	93	10
MUL	DirectMem	DirectMem	Register	160	8
MUL	DirectMem	Register	DirectMem	160	8
MUL	DirectMem	DirectMem	DirectMem	171	9
MUL	DirectMem	IndirectMem	Constant	100	11
MUL	DirectMem	Constant	IndirectMem	96	10
MUL	DirectMem	IndirectMem	Register	163	9
MUL	DirectMem	Register	IndirectMem	163	9
MUL	DirectMem	IndirectMem	DirectMem	174	10
MUL	DirectMem	DirectMem	IndirectMem	174	10
MUL	DirectMem	IndirectMem	IndirectMem	177	11
DIV	Register	Constant	Constant	4	3
DIV	Register	Constant	Register	89	7
DIV	Register	Register	Constant	92	6
DIV	Register	Register	Register	164	4
DIV	Register	DirectMem	Constant	103	7
DIV	Register	Constant	DirectMem	94	7
DIV	Register	DirectMem	Register	175	5
DIV	Register	Register	DirectMem	175	5
DIV	Register	DirectMem	DirectMem	186	6
DIV	Register	IndirectMem	Constant	106	8
DIV	Register	Constant	IndirectMem	97	7
DIV	Register	IndirectMem	Register	178	6
DIV	Register	Register	IndirectMem	178	6
DIV	Register	IndirectMem	DirectMem	189	7
DIV	Register	DirectMem	IndirectMem	189	7
DIV	Register	IndirectMem	IndirectMem	192	8

DIV	DirectMem	Constant	Constant	15	5
DIV	DirectMem	Constant	Register	103	10
DIV	DirectMem	Register	Constant	106	9
DIV	DirectMem	Register	Register	178	7
DIV	DirectMem	DirectMem	Constant	117	10
DIV	DirectMem	Constant	DirectMem	108	10
DIV	DirectMem	DirectMem	Register	189	8
DIV	DirectMem	Register	DirectMem	189	8
DIV	DirectMem	DirectMem	DirectMem	200	9
DIV	DirectMem	IndirectMem	Constant	120	11
DIV	DirectMem	Constant	IndirectMem	111	10
DIV	DirectMem	IndirectMem	Register	192	9
DIV	DirectMem	Register	IndirectMem	192	9
DIV	DirectMem	IndirectMem	DirectMem	203	10
DIV	DirectMem	DirectMem	IndirectMem	203	10
DIV	DirectMem	IndirectMem	IndirectMem	206	11
CMP	Register	Constant	Constant	4	3
CMP	Register	Constant	Register	7	5
CMP	Register	Register	Constant	6	6
CMP	Register	Register	Register	5	4
CMP	Register	DirectMem	Constant	17	7
CMP	Register	Constant	DirectMem	18	6
CMP	Register	DirectMem	Register	16	5
CMP	Register	Register	DirectMem	16	5
CMP	Register	DirectMem	DirectMem	27	6
CMP	Register	IndirectMem	Constant	20	8
CMP	Register	Constant	IndirectMem	21	7
CMP	Register	IndirectMem	Register	19	6
CMP	Register	Register	IndirectMem	19	6
CMP	Register	IndirectMem	DirectMem	30	7
CMP	Register	DirectMem	IndirectMem	30	7
CMP	Register	IndirectMem	IndirectMem	33	8
CMP	DirectMem	Constant	Constant	15	5
CMP	DirectMem	Constant	Register	21	8
CMP	DirectMem	Register	Constant	20	9
CMP	DirectMem	Register	Register	19	7
CMP	DirectMem	DirectMem	Constant	31	10
CMP	DirectMem	Constant	DirectMem	32	9
CMP	DirectMem	DirectMem	Register	30	8
CMP	DirectMem	Register	DirectMem	30	8
CMP	DirectMem	DirectMem	DirectMem	41	9
CMP	DirectMem	IndirectMem	Constant	34	11
CMP	DirectMem	Constant	IndirectMem	35	10
CMP	DirectMem	IndirectMem	Register	33	9
CMP	DirectMem	Register	IndirectMem	33	9
CMP	DirectMem	IndirectMem	DirectMem	44	10
CMP	DirectMem	DirectMem	IndirectMem	44	10
CMP	DirectMem	IndirectMem	IndirectMem	47	11

MOV	Register	Empty	Constant	4	3
MOV	Register	Empty	Register	2	2
MOV	Register	Empty	DirectMem	13	3
MOV	Register	Empty	IndirectMem	16	4
MOV	DirectMem	Empty	Constant	15	5
MOV	DirectMem	Empty	Register	14	3
MOV	DirectMem	Empty	DirectMem	27	6
MOV	DirectMem	Empty	IndirectMem	27	6
MOV	IndirectMem	Empty	Constant	18	6
MOV	IndirectMem	Empty	Register	17	4
MOV	IndirectMem	Empty	DirectMem	30	7
MOV	IndirectMem	Empty	IndirectMem	33	8
NOP	Empty	Empty	Empty	3	1
CJUMP	Empty	Empty	Empty	16	2
JUMP	Empty	Empty	Empty	24	4
RET	Empty	Empty	Empty	17	3
CALL	Empty	Empty	Empty	37	4

```
## Anything after '#' are comments.
## This is the technology file for 80286 processor.
## DirectMem means direct memory addressing.
## IndirectMem means indirect memory addressing.
#  OP    DESTINATION    SOURCE1     SOURCE2     time(clock cycles)  size(bytes)
   ALU     Register     Constant     Constant              2             4
   ALU     Register     Constant     Register              4             6
   ALU     Register     Register     Constant              5             6
   ALU     Register     Register     Register              4             4
   ALU     Register    DirectMem     Constant              8             7
   ALU     Register     Constant    DirectMem              9             7
   ALU     Register    DirectMem     Register              7             5
   ALU     Register     Register    DirectMem              9             5
   ALU     Register    DirectMem    DirectMem             12             6
   ALU     Register  IndirectMem     Constant              9             8
   ALU     Register     Constant  IndirectMem             10             8
   ALU     Register  IndirectMem     Register              8             6
   ALU     Register     Register  IndirectMem             10             6
   ALU     Register  IndirectMem    DirectMem             13             7
   ALU     Register    DirectMem  IndirectMem             13             7
   ALU     Register  IndirectMem  IndirectMem             14             8
   ALU    DirectMem     Constant     Constant              3             5
   ALU    DirectMem     Constant     Register              7             9
   ALU    DirectMem     Register     Constant              8             9
   ALU    DirectMem     Register     Register              7             7
   ALU    DirectMem    DirectMem     Constant             11            10
   ALU    DirectMem     Constant    DirectMem             12            10
   ALU    DirectMem    DirectMem     Register             10             8
   ALU    DirectMem     Register    DirectMem             12             8
   ALU    DirectMem    DirectMem    DirectMem             15             9
   ALU    DirectMem  IndirectMem     Constant             12            11
   ALU    DirectMem     Constant  IndirectMem             13            11
   ALU    DirectMem  IndirectMem     Register             11             9
   ALU    DirectMem     Register  IndirectMem             13             9
   ALU    DirectMem  IndirectMem    DirectMem             16            10
   ALU    DirectMem    DirectMem  IndirectMem             16            10
   ALU    DirectMem  IndirectMem  IndirectMem             17            11
   ALU     Register        Empty     Constant              2             4
   ALU     Register        Empty     Register              4             4
   ALU     Register        Empty    DirectMem              7             5
   ALU     Register        Empty  IndirectMem              8             6
   ALU    DirectMem        Empty     Constant              7             9
   ALU    DirectMem        Empty     Register              7             7
   ALU    DirectMem        Empty    DirectMem             10             8
   ALU    DirectMem        Empty  IndirectMem             11             9
   MUL     Register     Constant     Constant              2             4
   MUL     Register     Constant     Register             14             8
   MUL     Register     Register     Constant             14             6
```

MUL	Register	Register	Register	23	4
MUL	Register	DirectMem	Constant	17	7
MUL	Register	Constant	DirectMem	14	8
MUL	Register	DirectMem	Register	26	5
MUL	Register	Register	DirectMem	26	5
MUL	Register	DirectMem	DirectMem	29	6
MUL	Register	IndirectMem	Constant	18	8
MUL	Register	Constant	IndirectMem	15	8
MUL	Register	IndirectMem	Register	27	6
MUL	Register	Register	IndirectMem	27	6
MUL	Register	IndirectMem	DirectMem	30	7
MUL	Register	DirectMem	IndirectMem	30	7
MUL	Register	IndirectMem	IndirectMem	31	8
MUL	DirectMem	Constant	Constant	3	5
MUL	DirectMem	Constant	Register	17	11
MUL	DirectMem	Register	Constant	17	9
MUL	DirectMem	Register	Register	26	7
MUL	DirectMem	DirectMem	Constant	20	10
MUL	DirectMem	Constant	DirectMem	17	11
MUL	DirectMem	DirectMem	Register	29	8
MUL	DirectMem	Register	DirectMem	29	8
MUL	DirectMem	DirectMem	DirectMem	32	9
MUL	DirectMem	IndirectMem	Constant	21	11
MUL	DirectMem	Constant	IndirectMem	18	11
MUL	DirectMem	IndirectMem	Register	30	9
MUL	DirectMem	Register	IndirectMem	30	9
MUL	DirectMem	IndirectMem	DirectMem	33	10
MUL	DirectMem	DirectMem	IndirectMem	33	10
MUL	DirectMem	IndirectMem	IndirectMem	34	11
DIV	Register	Constant	Constant	2	4
DIV	Register	Constant	Register	15	8
DIV	Register	Register	Constant	15	6
DIV	Register	Register	Register	24	4
DIV	Register	DirectMem	Constant	18	7
DIV	Register	Constant	DirectMem	15	8
DIV	Register	DirectMem	Register	27	5
DIV	Register	Register	DirectMem	27	5
DIV	Register	DirectMem	DirectMem	30	6
DIV	Register	IndirectMem	Constant	19	8
DIV	Register	Constant	IndirectMem	16	8
DIV	Register	IndirectMem	Register	28	6
DIV	Register	Register	IndirectMem	28	6
DIV	Register	IndirectMem	DirectMem	31	7
DIV	Register	DirectMem	IndirectMem	31	7
DIV	Register	IndirectMem	IndirectMem	32	8
DIV	DirectMem	Constant	Constant	3	5
DIV	DirectMem	Constant	Register	18	11
DIV	DirectMem	Register	Constant	18	9

DIV	DirectMem	Register	Register	27	7
DIV	DirectMem	DirectMem	Constant	21	10
DIV	DirectMem	Constant	DirectMem	18	11
DIV	DirectMem	DirectMem	Register	30	8
DIV	DirectMem	Register	DirectMem	30	8
DIV	DirectMem	DirectMem	DirectMem	33	9
DIV	DirectMem	IndirectMem	Constant	22	11
DIV	DirectMem	Constant	IndirectMem	19	11
DIV	DirectMem	IndirectMem	Register	31	9
DIV	DirectMem	Register	IndirectMem	31	9
DIV	DirectMem	IndirectMem	DirectMem	34	10
DIV	DirectMem	DirectMem	IndirectMem	34	10
DIV	DirectMem	IndirectMem	IndirectMem	35	11
CMP	Register	Constant	Constant	2	4
CMP	Register	Constant	Register	4	6
CMP	Register	Register	Constant	5	6
CMP	Register	Register	Register	4	4
CMP	Register	DirectMem	Constant	8	7
CMP	Register	Constant	DirectMem	8	7
CMP	Register	DirectMem	Register	7	5
CMP	Register	Register	DirectMem	8	5
CMP	Register	DirectMem	DirectMem	11	6
CMP	Register	IndirectMem	Constant	9	8
CMP	Register	Constant	IndirectMem	9	8
CMP	Register	IndirectMem	Register	8	6
CMP	Register	Register	IndirectMem	9	6
CMP	Register	IndirectMem	DirectMem	12	7
CMP	Register	DirectMem	IndirectMem	12	7
CMP	Register	IndirectMem	IndirectMem	13	8
CMP	DirectMem	Constant	Constant	3	5
CMP	DirectMem	Constant	Register	7	9
CMP	DirectMem	Register	Constant	8	9
CMP	DirectMem	Register	Register	7	7
CMP	DirectMem	DirectMem	Constant	11	10
CMP	DirectMem	Constant	DirectMem	11	10
CMP	DirectMem	DirectMem	Register	10	8
CMP	DirectMem	Register	DirectMem	11	8
CMP	DirectMem	DirectMem	DirectMem	14	9
CMP	DirectMem	IndirectMem	Constant	12	11
CMP	DirectMem	Constant	IndirectMem	12	11
CMP	DirectMem	IndirectMem	Register	11	9
CMP	DirectMem	Register	IndirectMem	12	9
CMP	DirectMem	IndirectMem	DirectMem	15	10
CMP	DirectMem	DirectMem	IndirectMem	15	10
CMP	DirectMem	IndirectMem	IndirectMem	16	11
MOV	Register	Empty	Constant	2	4
MOV	Register	Empty	Register	2	2
MOV	Register	Empty	DirectMem	5	3

MOV	Register	Empty	IndirectMem	6	4
MOV	DirectMem	Empty	Constant	3	5
MOV	DirectMem	Empty	Register	3	3
MOV	DirectMem	Empty	DirectMem	8	6
MOV	DirectMem	Empty	IndirectMem	8	6
MOV	IndirectMem	Empty	Constant	4	6
MOV	IndirectMem	Empty	Register	4	4
MOV	IndirectMem	Empty	DirectMem	9	7
MOV	IndirectMem	Empty	IndirectMem	10	8
NOP	Empty	Empty	Empty	2	1
CJUMP	Empty	Empty	Empty	12	2
JUMP	Empty	Empty	Empty	15	4
RET	Empty	Empty	Empty	15	3
CALL	Empty	Empty	Empty	18	4

```
## Anything after '#' are comments.
## This is the technology file for 68000 processor.
## DirectMem means direct memory addressing.
## IndirectMem means indirect memory addressing.
#  OP    DESTINATION    SOURCE1     SOURCE2     time(clock cycles)  size(bytes)
   ALU     Register     Constant     Constant                12            2
   ALU     Register     Constant     Register                18            4
   ALU     Register     Register     Constant                10            6
   ALU     Register     Register     Register                10            4
   ALU     Register    DirectMem     Constant                18            6
   ALU     Register     Constant    DirectMem                20            4
   ALU     Register    DirectMem     Register                18            4
   ALU     Register     Register    DirectMem                12            4
   ALU     Register    DirectMem    DirectMem                20            4
   ALU     Register  IndirectMem     Constant                26            8
   ALU     Register     Constant  IndirectMem                28            6
   ALU     Register  IndirectMem     Register                26            6
   ALU     Register     Register  IndirectMem                20            6
   ALU     Register  IndirectMem    DirectMem                28            6
   ALU     Register    DirectMem  IndirectMem                28            6
   ALU     Register  IndirectMem  IndirectMem                36            8
   ALU    DirectMem     Constant     Constant                20            4
   ALU    DirectMem     Constant     Register                30            6
   ALU    DirectMem     Register     Constant                22            8
   ALU    DirectMem     Register     Register                22            6
   ALU    DirectMem    DirectMem     Constant                30            8
   ALU    DirectMem     Constant    DirectMem                32            6
   ALU    DirectMem    DirectMem     Register                30            6
   ALU    DirectMem     Register    DirectMem                24            6
   ALU    DirectMem    DirectMem    DirectMem                32            6
   ALU    DirectMem  IndirectMem     Constant                38           10
   ALU    DirectMem     Constant  IndirectMem                40            8
   ALU    DirectMem  IndirectMem     Register                38            8
   ALU    DirectMem     Register  IndirectMem                32            8
   ALU    DirectMem  IndirectMem    DirectMem                40            8
   ALU    DirectMem    DirectMem  IndirectMem                40            8
   ALU    DirectMem  IndirectMem  IndirectMem                48           10
   ALU     Register        Empty     Constant                12            2
   ALU     Register        Empty     Register                10            4
   ALU     Register        Empty    DirectMem                18            4
   ALU     Register        Empty  IndirectMem                26            6
   ALU    DirectMem        Empty     Constant                30            6
   ALU    DirectMem        Empty     Register                22            6
   ALU    DirectMem        Empty    DirectMem                30            6
   ALU    DirectMem        Empty  IndirectMem                38            8
   MUL     Register     Constant     Constant                12            2
   MUL     Register     Constant     Register                47            6
   MUL     Register     Register     Constant                39            6
```

MUL	Register	Register	Register	74	4
MUL	Register	DirectMem	Constant	47	6
MUL	Register	Constant	DirectMem	47	6
MUL	Register	DirectMem	Register	82	4
MUL	Register	Register	DirectMem	74	4
MUL	Register	DirectMem	DirectMem	82	4
MUL	Register	IndirectMem	Constant	55	8
MUL	Register	Constant	IndirectMem	55	6
MUL	Register	IndirectMem	Register	90	6
MUL	Register	Register	IndirectMem	82	6
MUL	Register	IndirectMem	DirectMem	90	6
MUL	Register	DirectMem	IndirectMem	90	6
MUL	Register	IndirectMem	IndirectMem	98	8
MUL	DirectMem	Constant	Constant	20	4
MUL	DirectMem	Constant	Register	59	8
MUL	DirectMem	Register	Constant	51	8
MUL	DirectMem	Register	Register	86	6
MUL	DirectMem	DirectMem	Constant	59	8
MUL	DirectMem	Constant	DirectMem	59	8
MUL	DirectMem	DirectMem	Register	94	6
MUL	DirectMem	Register	DirectMem	86	6
MUL	DirectMem	DirectMem	DirectMem	94	6
MUL	DirectMem	IndirectMem	Constant	67	10
MUL	DirectMem	Constant	IndirectMem	67	8
MUL	DirectMem	IndirectMem	Register	102	8
MUL	DirectMem	Register	IndirectMem	94	8
MUL	DirectMem	IndirectMem	DirectMem	102	8
MUL	DirectMem	DirectMem	IndirectMem	102	8
MUL	DirectMem	IndirectMem	IndirectMem	110	10
DIV	Register	Constant	Constant	12	2
DIV	Register	Constant	Register	92	6
DIV	Register	Register	Constant	84	6
DIV	Register	Register	Register	162	4
DIV	Register	DirectMem	Constant	92	6
DIV	Register	Constant	DirectMem	92	6
DIV	Register	DirectMem	Register	170	4
DIV	Register	Register	DirectMem	162	4
DIV	Register	DirectMem	DirectMem	170	4
DIV	Register	IndirectMem	Constant	100	8
DIV	Register	Constant	IndirectMem	100	6
DIV	Register	IndirectMem	Register	178	6
DIV	Register	Register	IndirectMem	170	6
DIV	Register	IndirectMem	DirectMem	178	6
DIV	Register	DirectMem	IndirectMem	178	6
DIV	Register	IndirectMem	IndirectMem	186	8
DIV	DirectMem	Constant	Constant	20	4
DIV	DirectMem	Constant	Register	104	8
DIV	DirectMem	Register	Constant	96	8

DIV	DirectMem	Register	Register	174	6
DIV	DirectMem	DirectMem	Constant	104	8
DIV	DirectMem	Constant	DirectMem	104	8
DIV	DirectMem	DirectMem	Register	182	6
DIV	DirectMem	Register	DirectMem	174	6
DIV	DirectMem	DirectMem	DirectMem	182	6
DIV	DirectMem	IndirectMem	Constant	112	10
DIV	DirectMem	Constant	IndirectMem	112	8
DIV	DirectMem	IndirectMem	Register	190	8
DIV	DirectMem	Register	IndirectMem	182	8
DIV	DirectMem	IndirectMem	DirectMem	190	8
DIV	DirectMem	DirectMem	IndirectMem	190	8
DIV	DirectMem	IndirectMem	IndirectMem	198	10
CMP	Register	Constant	Constant	12	2
CMP	Register	Constant	Register	16	4
CMP	Register	Register	Constant	10	6
CMP	Register	Register	Register	8	4
CMP	Register	DirectMem	Constant	18	6
CMP	Register	Constant	DirectMem	18	4
CMP	Register	DirectMem	Register	16	4
CMP	Register	Register	DirectMem	10	4
CMP	Register	DirectMem	DirectMem	18	4
CMP	Register	IndirectMem	Constant	26	8
CMP	Register	Constant	IndirectMem	26	6
CMP	Register	IndirectMem	Register	24	6
CMP	Register	Register	IndirectMem	18	6
CMP	Register	IndirectMem	DirectMem	26	6
CMP	Register	DirectMem	IndirectMem	26	6
CMP	Register	IndirectMem	IndirectMem	34	8
CMP	DirectMem	Constant	Constant	20	4
CMP	DirectMem	Constant	Register	28	6
CMP	DirectMem	Register	Constant	22	8
CMP	DirectMem	Register	Register	20	6
CMP	DirectMem	DirectMem	Constant	30	8
CMP	DirectMem	Constant	DirectMem	30	6
CMP	DirectMem	DirectMem	Register	28	6
CMP	DirectMem	Register	DirectMem	22	6
CMP	DirectMem	DirectMem	DirectMem	30	6
CMP	DirectMem	IndirectMem	Constant	38	10
CMP	DirectMem	Constant	IndirectMem	38	8
CMP	DirectMem	IndirectMem	Register	36	8
CMP	DirectMem	Register	IndirectMem	30	8
CMP	DirectMem	IndirectMem	DirectMem	38	8
CMP	DirectMem	DirectMem	IndirectMem	38	8
CMP	DirectMem	IndirectMem	IndirectMem	46	10
MOV	Register	Empty	Constant	12	2
MOV	Register	Empty	Register	4	2
MOV	Register	Empty	DirectMem	12	2

MOV	Register	Empty	IndirectMem	20	4
MOV	DirectMem	Empty	Constant	20	4
MOV	DirectMem	Empty	Register	12	2
MOV	DirectMem	Empty	DirectMem	24	4
MOV	DirectMem	Empty	IndirectMem	24	4
MOV	IndirectMem	Empty	Constant	28	6
MOV	IndirectMem	Empty	Register	20	4
MOV	IndirectMem	Empty	DirectMem	32	6
MOV	IndirectMem	Empty	IndirectMem	40	8
NOP	Empty	Empty	Empty	4	2
CJUMP	Empty	Empty	Empty	18	2
JUMP	Empty	Empty	Empty	12	4
RET	Empty	Empty	Empty	20	4
CALL	Empty	Empty	Empty	48	4

```
## Anything after '#' are comments.
## This is the technology file for 68020 processor.
## DirectMem means direct memory addressing.
## IndirectMem means indirect memory addressing.
#  OP    DESTINATION    SOURCE1     SOURCE2     time(clock cycles)  size(bytes)
   ALU     Register     Constant     Constant                 6            2
   ALU     Register     Constant     Register                 9            4
   ALU     Register     Register     Constant                 5            6
   ALU     Register     Register     Register                 5            4
   ALU     Register    DirectMem     Constant                 9            6
   ALU     Register     Constant    DirectMem                10            4
   ALU     Register    DirectMem     Register                 9            4
   ALU     Register     Register    DirectMem                 6            4
   ALU     Register    DirectMem    DirectMem                10            4
   ALU     Register  IndirectMem     Constant                13            8
   ALU     Register     Constant  IndirectMem                14            6
   ALU     Register  IndirectMem     Register                13            6
   ALU     Register     Register  IndirectMem                10            6
   ALU     Register  IndirectMem    DirectMem                14            6
   ALU     Register    DirectMem  IndirectMem                14            6
   ALU     Register  IndirectMem  IndirectMem                18            8
   ALU    DirectMem     Constant     Constant                 8            4
   ALU    DirectMem     Constant     Register                15            6
   ALU    DirectMem     Register     Constant                11            8
   ALU    DirectMem     Register     Register                11            6
   ALU    DirectMem    DirectMem     Constant                15            8
   ALU    DirectMem     Constant    DirectMem                16            6
   ALU    DirectMem    DirectMem     Register                15            6
   ALU    DirectMem     Register    DirectMem                12            6
   ALU    DirectMem    DirectMem    DirectMem                16            6
   ALU    DirectMem  IndirectMem     Constant                19           10
   ALU    DirectMem     Constant  IndirectMem                20            8
   ALU    DirectMem  IndirectMem     Register                19            8
   ALU    DirectMem     Register  IndirectMem                16            8
   ALU    DirectMem  IndirectMem    DirectMem                20            8
   ALU    DirectMem    DirectMem  IndirectMem                20            8
   ALU    DirectMem  IndirectMem  IndirectMem                24           10
   ALU     Register        Empty     Constant                 6            2
   ALU     Register        Empty     Register                 5            4
   ALU     Register        Empty    DirectMem                 9            4
   ALU     Register        Empty  IndirectMem                13            6
   ALU    DirectMem        Empty     Constant                15            6
   ALU    DirectMem        Empty     Register                11            6
   ALU    DirectMem        Empty    DirectMem                15            6
   ALU    DirectMem        Empty  IndirectMem                19            8
   MUL     Register     Constant     Constant                 6            2
   MUL     Register     Constant     Register                28            6
   MUL     Register     Register     Constant                24            6
```

MUL	Register	Register	Register	46	4
MUL	Register	DirectMem	Constant	28	6
MUL	Register	Constant	DirectMem	28	6
MUL	Register	DirectMem	Register	50	4
MUL	Register	Register	DirectMem	46	4
MUL	Register	DirectMem	DirectMem	50	4
MUL	Register	IndirectMem	Constant	32	8
MUL	Register	Constant	IndirectMem	32	6
MUL	Register	IndirectMem	Register	54	6
MUL	Register	Register	IndirectMem	50	6
MUL	Register	IndirectMem	DirectMem	54	6
MUL	Register	DirectMem	IndirectMem	54	6
MUL	Register	IndirectMem	IndirectMem	58	8
MUL	DirectMem	Constant	Constant	8	4
MUL	DirectMem	Constant	Register	34	8
MUL	DirectMem	Register	Constant	30	8
MUL	DirectMem	Register	Register	52	6
MUL	DirectMem	DirectMem	Constant	34	8
MUL	DirectMem	Constant	DirectMem	34	8
MUL	DirectMem	DirectMem	Register	56	6
MUL	DirectMem	Register	DirectMem	52	6
MUL	DirectMem	DirectMem	DirectMem	56	6
MUL	DirectMem	IndirectMem	Constant	38	10
MUL	DirectMem	Constant	IndirectMem	38	8
MUL	DirectMem	IndirectMem	Register	60	8
MUL	DirectMem	Register	IndirectMem	56	8
MUL	DirectMem	IndirectMem	DirectMem	60	8
MUL	DirectMem	DirectMem	IndirectMem	60	8
MUL	DirectMem	IndirectMem	IndirectMem	64	10
DIV	Register	Constant	Constant	6	2
DIV	Register	Constant	Register	45	6
DIV	Register	Register	Constant	41	6
DIV	Register	Register	Register	80	4
DIV	Register	DirectMem	Constant	45	6
DIV	Register	Constant	DirectMem	45	6
DIV	Register	DirectMem	Register	84	4
DIV	Register	Register	DirectMem	80	4
DIV	Register	DirectMem	DirectMem	84	4
DIV	Register	IndirectMem	Constant	49	8
DIV	Register	Constant	IndirectMem	49	6
DIV	Register	IndirectMem	Register	88	6
DIV	Register	Register	IndirectMem	84	6
DIV	Register	IndirectMem	DirectMem	88	6
DIV	Register	DirectMem	IndirectMem	88	6
DIV	Register	IndirectMem	IndirectMem	92	8
DIV	DirectMem	Constant	Constant	8	4
DIV	DirectMem	Constant	Register	51	8
DIV	DirectMem	Register	Constant	47	8

DIV	DirectMem	Register	Register	86	6
DIV	DirectMem	DirectMem	Constant	51	8
DIV	DirectMem	Constant	DirectMem	51	8
DIV	DirectMem	DirectMem	Register	90	6
DIV	DirectMem	Register	DirectMem	86	6
DIV	DirectMem	DirectMem	DirectMem	90	6
DIV	DirectMem	IndirectMem	Constant	55	10
DIV	DirectMem	Constant	IndirectMem	55	8
DIV	DirectMem	IndirectMem	Register	94	8
DIV	DirectMem	Register	IndirectMem	90	8
DIV	DirectMem	IndirectMem	DirectMem	94	8
DIV	DirectMem	DirectMem	IndirectMem	94	8
DIV	DirectMem	IndirectMem	IndirectMem	98	10
CMP	Register	Constant	Constant	6	2
CMP	Register	Constant	Register	9	4
CMP	Register	Register	Constant	6	6
CMP	Register	Register	Register	5	4
CMP	Register	DirectMem	Constant	10	6
CMP	Register	Constant	DirectMem	10	4
CMP	Register	DirectMem	Register	9	4
CMP	Register	Register	DirectMem	6	4
CMP	Register	DirectMem	DirectMem	10	4
CMP	Register	IndirectMem	Constant	14	8
CMP	Register	Constant	IndirectMem	14	6
CMP	Register	IndirectMem	Register	13	6
CMP	Register	Register	IndirectMem	10	6
CMP	Register	IndirectMem	DirectMem	14	6
CMP	Register	DirectMem	IndirectMem	14	6
CMP	Register	IndirectMem	IndirectMem	18	8
CMP	DirectMem	Constant	Constant	8	4
CMP	DirectMem	Constant	Register	15	6
CMP	DirectMem	Register	Constant	12	8
CMP	DirectMem	Register	Register	11	6
CMP	DirectMem	DirectMem	Constant	16	8
CMP	DirectMem	Constant	DirectMem	16	6
CMP	DirectMem	DirectMem	Register	15	6
CMP	DirectMem	Register	DirectMem	12	6
CMP	DirectMem	DirectMem	DirectMem	16	6
CMP	DirectMem	IndirectMem	Constant	20	10
CMP	DirectMem	Constant	IndirectMem	20	8
CMP	DirectMem	IndirectMem	Register	19	8
CMP	DirectMem	Register	IndirectMem	16	8
CMP	DirectMem	IndirectMem	DirectMem	20	8
CMP	DirectMem	DirectMem	IndirectMem	20	8
CMP	DirectMem	IndirectMem	IndirectMem	24	10
MOV	Register	Empty	Constant	6	2
MOV	Register	Empty	Register	2	2
MOV	Register	Empty	DirectMem	6	2

MOV	Register	Empty	IndirectMem	10	4
MOV	DirectMem	Empty	Constant	8	4
MOV	DirectMem	Empty	Register	6	2
MOV	DirectMem	Empty	DirectMem	12	4
MOV	DirectMem	Empty	IndirectMem	12	4
MOV	IndirectMem	Empty	Constant	12	6
MOV	IndirectMem	Empty	Register	10	4
MOV	IndirectMem	Empty	DirectMem	16	6
MOV	IndirectMem	Empty	IndirectMem	20	8
NOP	Empty	Empty	Empty	2	1
CJUMP	Empty	Empty	Empty	12	2
JUMP	Empty	Empty	Empty	7	4
RET	Empty	Empty	Empty	10	2
CALL	Empty	Empty	Empty	28	4

B SpecChart Description of the Medical System

```
behavior SYS type sequential substates is
   use bit_functions;
   port x_step: in BIT_VECTOR (7 downto 0) ;
   port y_step: in BIT_VECTOR (7 downto 0) ;
   port D: in INTEGER ;
   port start1: in BIT ;
   port ready: in BIT ;
   port x_pos: out BIT ;
   port y_pos: out BIT ;
   port x_dir: out BIT ;
   port y_dir: out BIT ;
   port motor_reset: out BIT ;
   port output_p: out BIT_VECTOR (15 downto 0) ;
   port strobe: out BIT ;
   port diskport: out INTEGER ;
   type MEMORY is array (0 to 1000) of integer;
   variable x_step_var: BIT_VECTOR (7 downto 0);
   variable y_step_var: BIT_VECTOR (7 downto 0);
   variable x_count: INTEGER;
   variable y_count: INTEGER;
   variable x_dir_var: BIT;
   variable y_dir_var: BIT;
   variable M: MEMORY;
   variable found: BIT;
   variable index: INTEGER;
   variable volume: INTEGER;
   variable num_data: INTEGER;
   variable flat_level: INTEGER;
   variable anterior_wall: INTEGER;
   variable posterior_wall: INTEGER;
begin

   INITIATE : (EOC, true, EMPTY1);
   EMPTY1 : (EOC, y_count < B2I(y_step_var), MOTOR_CONTROL1),
(EOC, y_count >= B2I(y_step_var), OUTPUT);
   MOTOR_CONTROL1 : (EOC, true, EMPTY2);
   OUTPUT : (EOC, true, STOP);
   EMPTY2 : (EOC, x_count < B2I(x_step_var), XBODY),
(EOC, x_count >= B2I(x_step_var), INCY);
   INCY : (EOC, true, EMPTY1);
   XBODY : (EOC, true, EMPTY2);

   behavior INITIATE type code is
   begin
      wait until (start1 = '1');
```

```
   x_step_var := x_step;
   y_step_var := y_step;
   motor_reset <=  '1';
   wait for 10 ns;
   Motor_reset <=  '0';
   y_count := 0;
   x_count := 0;
   y_pos <=  '1';
   wait for 10 ns;
   y_pos <=  '0';
   x_dir_var := '0';
   y_dir_var := '0';
   num_data := 1000;
   volume := 0;
   wait for 10 ns;
end INITIATE;

behavior EMPTY1 type code is
begin
   null;
end EMPTY1;

behavior MOTOR_CONTROL1 type code is
begin
   x_count := 0;
   x_pos <=  '1';
   y_pos <=  '1';
   x_dir <=  x_dir_var;
   y_dir <=  y_dir_var;
   wait for 1 ns;
   x_pos <=  '0';
   y_pos <=  '0';
   wait for 1 ns;
   if (x_dir_var = '0') then
      x_dir_var := '1';
   else
      x_dir_var := '0';
   end if;
end MOTOR_CONTROL1;

behavior OUTPUT type code is
begin
   output_p <=  I2B(volume,16);
   wait for 1 ns;
end OUTPUT;

behavior EMPTY2 type code is
begin
```

```
   null;
end EMPTY2;

behavior INCY type code is
begin
   y_count := y_count + 1;
end INCY;

behavior XBODY type sequential substates is
begin
   MOTOR_CONTROL2 : (EOC, true, DATA_ACQUISITION);
   DATA_ACQUISITION : (EOC, true, FLATLEVEL);
   FLATLEVEL : (EOC, true, ANTERIORWALL);
   ANTERIORWALL : (EOC, true, POSTERIORWALL);
   POSTERIORWALL : (EOC, true, COMPUTE_VOLUME);
   COMPUTE_VOLUME : (EOC, true, DISK);
   DISK : (EOC, true, INCX);
   INCX : (EOC, true, STOP);

   behavior MOTOR_CONTROL2 type code is
   begin
      x_pos <=  '1';
      wait for 1 ns;
      x_pos <=  '0';
      wait for 1 ns;
   end MOTOR_CONTROL2;

   behavior DATA_ACQUISITION type code is
      variable temp_index: INTEGER;
   begin
      temp_index := 0;
      while (temp_index < num_data) loop
         strobe <=  '1';
         wait until (ready = '1');
         strobe <=  '0';
         M(temp_index) := D;
         wait until (ready = '0');
         temp_index := temp_index + 1;
      end loop ;
   end DATA_ACQUISITION;

   behavior FLATLEVEL type code is
      variable temp1: INTEGER;
      variable temp2: INTEGER;
   begin
      temp2 := 0;
      for temp1 in (num_data - 2) to (num_data - 1) loop
         temp2 := temp2 + M(temp1);
```

```
      end loop ;
      flat_level := B2I(SHL0(I2B(temp2,8),3)) + 10;
   end FLATLEVEL;

   behavior ANTERIORWALL type code is
      variable temp3: INTEGER;
   begin
      found := '0';
      anterior_wall := 0;
      index := 0;
      while (found = '0') loop
         temp3 := M(index);
         if (temp3 > flat_level) then
            index := index + 1;
         else
            anterior_wall := index;
            found := '1';
         end if;
      end loop ;
   end ANTERIORWALL;

   behavior POSTERIORWALL type code is
      variable temp4: INTEGER;
   begin
      found := '0';
      posterior_wall := 0;
      while ((found = '0') and (index < num_data)) loop
         temp4 := M(index);
         if (temp4 < flat_level) then
            index := index + 1;
         else
            posterior_wall := index;
            found := '1';
         end if;
      end loop ;
   end POSTERIORWALL;

   behavior COMPUTE_VOLUME type code is
   begin
      if (found = '1') then
         volume := volume + (posterior_wall * posterior_wall *
posterior_wall - anterior_wall * anterior_wall
* anterior_wall);
      else
         volume := 0;
      end if;
   end COMPUTE_VOLUME;
```

```
      behavior DISK type code is
         variable data_index: INTEGER;
      begin
         data_index := 0;
         for data_index in 0 to (num_data - 1) loop
            diskport <=  M(data_index);
            wait for 1 ns;
         end loop ;
      end DISK;

      behavior INCX type code is
      begin
         x_count := x_count + 1;
      end INCX;
   end XBODY;
end SYS;
```