The System Architect's Workbench

D. E. Thomas, E. M. Dirkes, R. A. Walker,
J. V. Rajan, J. A. Nestor*, R. L. Blackburn

Electrical and Computer Engineering Department
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper presents an overview of The System Architect's Workbench, a behavioral synthesis system under development at Carnegie Mellon University. This system converts an abstract behavioral description of a piece of hardware into a set of register-transfer components and a control sequence table. Two synthesis methodologies are supported: one is tuned specifically to design microprocessors, and the other supports a more general design style. Results are presented here for both approaches.

Introduction

Recently, considerable research effort has been directed toward the synthesis of register transfer level designs from abstract behavioral descriptions [McFarland86, Pangrle86, Parker86, Paulin86]. Each of these efforts has defined a behavioral representation that describes only the function of a system to be built, much in the same way that a software program describes only a function. Synthesis research at this level has then focused on the specification of the control schedule and data path to implement the behavior. In these synthesis steps, operations in the behavior are assigned to control steps, registers are assigned to hold values, functional units are assigned to perform operations, and data path interconnection is generated to connect the registers and functional units. This paper presents results from the System Architect's Workbench [Walker87] research project (Figure 1), which is an example of such a synthesis system.

There is a wide range of applications for synthesis tools at this level, ranging from producing designs that are specific to an application to those that are more general purpose. In both of these areas there are a number of sub-classifications of designs, for instance, pipelined designs, interfaces, and signal processing designs. A successful synthesis system should be able to produce designs in all of these sub-areas, or design styles [Thomas81] as we shall call them.

This paper discusses two approaches to synthesis, and presents results from the two approaches. One approach suggests that the knowledge used in the synthesis of each of these design styles should be an integral part of the synthesis program. Thus, a style-specific synthesis program includes information about tradeoffs, and about specific techniques taken in the synthesis of a system of that style. This approach mimics the situation where designers become experts in the design of, say, microprocessors, but not also in a different style such as digital signal processing. One aspect of the System Architect's Workbench, the SUGAR program, follows the style-specific approach.

A second approach suggests that general synthesis algorithms can be used to produce results in a range of design styles. The general synthesis tool, based on this approach, is tunable on a per-design basis so that it can appropriately synthesize different design styles. Instead of directly encoding knowledge about the design style into the synthesis algorithms, a table driven or knowledge database approach is used where the values in the tables or database, along with information derived from analysis of the design, are used to guide the decision making. One path through the System Architect's Workbench follows the general approach; it consists of the behavioral transformations, the CSTEP program, and the APARTY/EMUCS/Busser programs.

The style-specific approach is interesting because it typically leads to fast synthesis programs that give good results for particular design styles. However, designs are not always stylistically homogeneous. For instance, all microprocessors must have an interface, but the design knowledge used in designing the interface and that used in designing the data path are quite different. The general approach to synthesis should be better able to synthesize a wide range of design styles, even though such approaches generally run more slowly. A long term research question is whether results from the general approach can overtake the results of the style-specific approach.

Although this paper does not give conclusive results indicating the appropriate model for synthesis systems, it does provide interesting comparisons based on running examples through design programs in the System Architect's Workbench that exhibit each approach.

---

*now with the Electrical and Computer Engineering Department, Illinois Institute of Technology, Chicago, IL 60616.

presented at The 25th Design Automation Conference

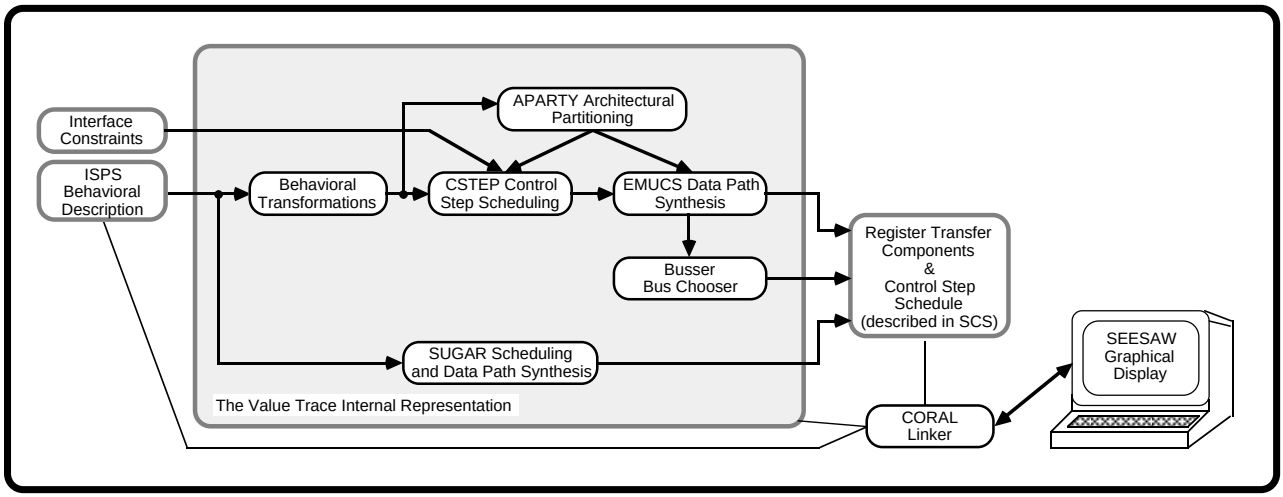

Figure 1
The System Architect's Workbench

Workbench Overview

Before beginning either of the synthesis paths in the Workbench, a behavioral description of the system to be designed is written in the ISPS language, and translated to a directed acyclic graph called the Value Trace [Snow 78], or VT, as shown in Figure 2. This Value Trace is then used as the internal behavioral representation for the Workbench. Procedures and labeled blocks in ISPS are mapped onto subgraphs called vtbodies, operators are mapped onto nodes, and carriers are mapped onto arcs. SELECT operators are used to perform IF / THEN and DECODEing operations; a SELECT has a set of branches, each containing a set of operators, and only one branch is active at a time. Figure 2 shows an example of an ISPS DECODE operation that is mapped onto a SELECT operator with two branches. If the value of IR is 9, the first branch is executed, and if the value is 41, the second branch is executed.

Figure 1 illustrates the design programs in the System Architect's Workbench. The general and style-specific approaches to synthesis are shown horizontally, operating on the VT representation and generating the register-transfer and control schedule information, represented in the SCS language [Vasantharajan82]. The CORAL program [Blackburn88] maintains the correlations between the initial ISPS description, the VT, and the synthesized design representation, providing a correlated basis for user interrogation, verification, and other applications. The SEESAW program [Blackburn88] is used to display the various representations graphically and to highlight the relationships between them.

The top rows of Figure 1 show the general approach to design. First, behavioral and structural transformations [Snow78, Walker87] are applied to the VT. Typically, this step transforms the VT from one that reflects a software engineered ISPS description into one more appropriate for implementation. Next, the CSTEP control step scheduling program [Nestor87] assigns operators in the VT to control steps using maximum and minimum timing constraints that may be specified between operations in the ISPS description. The EMUCS data path synthesis program [Hitchcock83, Nestor87] is used to complete the register-transfer level design. The Busser program chooses the busses to interconnect the datapath elements. Top-down information provided by the architectural partitioning program APARTY on how to partition the operations in the VT into separate logical groups is taken into account by CSTEP, EMUCS, and Busser. This part of the system has been used to design interface hardware, small microprocessors, and real-time controller hardware.

```
run {main} := BEGIN
  repeat BEGIN
    IR = M[PC] NEXT
    PC = PC + 1 NEXT
    DECODE IR => BEGIN
      9\ORA := BEGIN
        A = A OR M[PC] NEXT
        PC = PC + 1
        END,
      41\AND := BEGIN
        A = A AND M[PC] NEXT
        PC = PC + 1
        END
      END
    END
  END
```

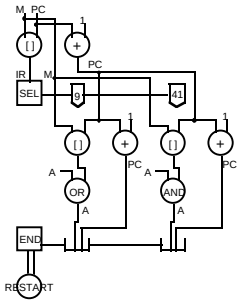

Figure 2
Translation from ISPS to the Value Trace

The lower row of Figure 1 shows the design-style-specific approach to design. The SUGAR program combines transformation, control step scheduling, and data path synthesis into a single program. Unlike the general synthesis approach, SUGAR is aimed at the synthesis of microprocessors, and specifies the control schedule based on the data path resources needed, rather than assigning the control schedule before data path allocation.

The rest of this paper is organized around the approaches taken by the behavioral transformations, APARTY, CSTEP, EMUCS, Busser and SUGAR, and the current results being obtained that highlight the capabilities of each system. Results for a common design example, the MCS6502, provide a means for comparing the style-specific and general synthesis approaches. A digital filter design is also shown as a result of the general approach.

The Transformations

Before control steps are scheduled and the data path allocated, behavioral and structural transformations can be applied. This set of transformations allows the designer to explore system-level design alternatives by interactively transforming the VT. The result can then be analyzed in terms of the control step schedule.

Some transformations serve primarily to eliminate biases in the designer's coding style. For example, a highly modular, easily readable description can be transformed to expand vtbodies inline and thus remove unnecessary subroutine calls (each of which might require a microsubroutine jump). Likewise, nested SELECTs can be combined, and operations can be moved into and out of SELECT branches, often achieving a better packing of operations into control steps. These transformations operate at about the same level as those in Flamel [Trickey87], although that system is more concerned with basic-block transformations and graph height reduction.

Other transformations serve to explore system-level design alternatives [Walker87], creating processes and pipestages, and structurally partitioning the design (perhaps in response to physical constraints). For example, the behavior can be partitioned into two concurrent processes, or it can be pipelined into two or more pipestages. This same behavior can also be structurally partitioned to split the design across two or more chips. These alternatives are shown in Figure 3.

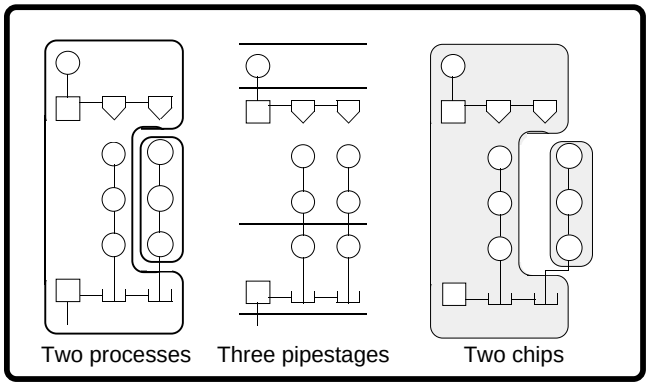

Figure 3
Alternatives in Transformation and Partitioning

An Example – The 6502

The following example shows the results of applying the transformations to a description of the MCS6502. Our initial ISPS description of the 6502 is highly modular; it contains many vtbodies and many CALLs, and three levels of instruction decoding. Since each CALL and each level of decoding requires a microsubroutine jump, 16 cycles are required to execute an ORA (OR accumulator) immediate instruction.

By recursively expanding all vtbodies inline, one large vtbody can be produced. Various SELECT transformations can then be used to reduce the original description's three levels of instruction decoding to one level of decoding; this produces the following unpipelined control step schedule. This listing shows a simulation trace of the ORA (OR Accumulator) immediate instruction; blank lines separate control steps produced by CSTEP:

```
v46.x1      IR = M[PC]
v46.x2      PC = PC + 1
v46.x3      SELECT (IR)

v46.x30     PC = PC + 1
v46.x31     GET1 = M[PC]
v46.x32     A = A OR GET1
v46.x33     x33.p1:SETNZ = A<7:7>
v46.x34     P<7:7> = x33.p1:SETNZ
v46.x35     x35.p1 = A EQL 0
v46.x36     P<1:1> = x35.p1
            IMMED = PC
            EADD1 = PC
            SETNZ = A

v46.x1629   RESTART @v46:RUN
```

This example will be discussed further in later sections of this paper, and will be referred to as the 6502 common design example.

Using the system-level transformations mentioned earlier, this design can also be pipelined, resulting in a CSTEP control schedule with the same control step division as the commercial MCS6502. Since EMUCS can not yet synthesize pipelined designs, no data path is available. Note that, unlike the unpiplined design, this design separates the instruction fetch and decoding into two different control steps. In the instruction trace shown below, the two stages are separated by a dashed line, and are assumed to be pipelined:

```
v46.x1      IR = M[PC]
v46.x2      PC = PC + 1
v46.x3      SEND {CALLv47}
v46.x4      RECV {LEAVEv47}
v46.x5      RESTART @v46:RUN

--------------------------------

v47.x1      RECV {CALLv47}
v47.x2      PC = PC + 1
v47.x3      GET1 = M[PC]
v47.x4      SELECT (IR)

v47.x30     A = A OR GET1
v47.x31     x31.p1:SETNZ = A<7:7>
v47.x32     P<7:7> = x31.p1:SETNZ
v47.x33     x33.p1 = A EQL 0
v47.x34     P<1:1> = x33.p1
v47.x35     SEND {LEAVEv47}
v47.x36     RESTART @v47:EXECUTE
```

Like the commercial MCS6502, this design is partitioned to require 3 cycles to execute the ORA immediate instruction, and to initiate a new instruction every 2 cycles. Beginning with this same "generic" behavioral description, other variations on the 6502, with a different choice of pipestages, or with a different control schedule, can also be explored.

Future work may address the problem of applying some of these transformations automatically. Since the unpipelined common design example can be characterized as a single-vtbody, single-level-of-decoding design, this should be relatively straightforward.

Pipelining, however, may remain interactive to allow the designer to explore many alternatives, rather than automatically attempting to choose a single "best" design.

Another Example – The Berkeley Risc-1

Another example that was transformed was the Berkeley Risc-1. This example was particularly interesting because the ISPS description was written outside our research group. Although it was necessary to rewrite a small portion of the description to eliminate multiple exits from procedures, to eliminate non-local exits from procedures, and to allow 32-bit memory accesses, the resulting design was transformed in much the same manner as the 6502. These transformations produced a 3-stage pipeline, and like the "blue" design for the original Berkeley Risc-1 [Fitzpatrick81], the control schedule for this design would execute an Add instruction in 3 cycles. Again, other variations on this design could be explored.

Control Step Scheduling

In the general synthesis path, after the design has been transformed, a control step schedule must be generated to sequence the design for EMUCS. The goal of this control step scheduling is to create an assignment of operations to control steps; each control step will correspond to a control state in the synthesized design, and will execute in one clock period. To obtain a correct design, this assignment must satisfy both the data flow and control flow dependencies in the VT.

The CSTEP Scheduling Algorithm

In general, the control step scheduling problem is very similar to microcode compaction [Davidson81]; early approaches have used heuristics taken directly from microcode compaction research such as *first-come, first-served (FCFS) scheduling* and *list scheduling*. In FCFS scheduling, operations are scheduled iteratively into successive control steps as allowed by data dependencies and their initial order. In list scheduling, operations are scheduled iteratively as allowed by data dependencies. The order in which operations are placed into a control step is determined by a heuristic *priority function* that is applied to all operations that have not yet been placed into a control step.

More recent approaches to scheduling have recognized the need to minimize resources and satisfy performance constraints expressed as maximum time constraints. Girczyc and Knight [Girczyc84] use "urgency scheduling", an approach similar to list scheduling that uses a priority function called an "urgency factor" that is related to whether delaying the placement of an operation will violate maximum time constraints. MAHA [Parker86] determines a critical path of operations based on maximum time constraints and schedules operations on the critical path first, scheduling operations off the critical path in a way that attempts to minimize hardware. HAL [Paulin87] uses force directed scheduling to balance hardware utilization between the allowed control steps.

The control step scheduler CSTEP uses a modified form of list scheduling to schedule not only data and control operations, but interface operations as well [Nestor87]. While most other schedulers considered only maximum time constraints, CSTEP considers both minimum and maximum time constraints. In addition, operations between time-constrained operations are scheduled independently, yielding greater flexibility in scheduling.

CSTEP schedules operations into control steps one basic block at a time. Basic blocks are scheduled in execution order using a depth-first traversal of the control flow graph. This ensures that control flow dependencies are satisfied. Data flow dependencies are guaranteed by only considering for placement those operators whose inputs are produced earlier in the schedule. For each basic block, operators are considered for placement in the "current" control step using a *priority function* that reflects whether placing the operator in the current step will violate a minimum time constraint, and whether placing an operator in a later step will violate a maximum time constraint. Hardware limits may be specified to limit the number of a type of operator scheduled in any one control step.

CSTEP Results

CSTEP is reasonably efficient; scheduling the 6502 common design example requires 12.34 CPU seconds on a DEC VAXstation II. The number of iterations of the algorithm in a basic block containing n operations grows in the worst case with complexity $O(n)$. The number of calls to the priority function grows in the worst case with complexity $O(n^2)$, but this can be improved by re-evaluating the priority of an operation only when conditions have changed from its previous evaluation.

Table 1 lists a set of instructions and execution times for the 6502; all times are measured in control steps. The CSTEP results assume an edge-triggered clocking scheme, the commercial design assumes a 2-phase clock, and the SUGAR results assume a 4-phase clock. The second column shows the instruction execution times from the CSTEP control schedule for the unpipelined 6502 common design example. The third and fourth columns of the table show instruction execution times for two directly comparable pipelined designs – the 6502 2-stage pipeline example, and the commercial MCS6502; the figures for these two columns assume that the instruction was prefetched at the same time as the previous instruction was executed. In the third column, when two times are given, the second is for indexing across a page boundary. The common design example and the pipelined designs are not directly comparable, since the common design example is unpipelined, and combines the instruction fetch and decoding into a single control step (which the other designs do not). However, the unpipelined results are still acceptable, and often within one cycle of the pipelined designs. The results shown in the fifth column will be discussed later in the section on SUGAR.

Table 1
6502 Instruction Execution Times

Instruction and Addressing Mode	Unpipelined	Comparable Pipelines		Unpipelined
	6502 Common Design Example (CSTEP)	Commercial MCS6502	6502 2-Stage Pipeline Example (CSTEP)	SUGAR 6502 Example
ORA, Immediate	3	2	2	4
ORA, Zero Page	4	3	3	4
ORA, Zero Page, X	4	4	3	5
ORA, Absolute	5	4	4	5
ORA, Absolute, X	5	4/5	4	7
ORA, Absolute, Y	5	4/5	4	7
ORA, (Indirect, X)	7	6	5	7
ORA, (Indirect), Y	7	5/6	5	8

Architectural Partitioning

In the general synthesis path of the Workbench, after behavioral transforms have been applied, the architectural partitioner APARTY can be run, and the partitioning results can be used to guide the data path synthesis tools: CSTEP, EMUCS, and Busser.

Architectural partitioning is the division of the functionality of an architecture between physical units. It is generally done to meet area and performance restrictions, but must also consider the interconnection of the physical units. There are several types of partitioning, including instruction set partitioning and intra-chip partitioning. Instruction set partitioning divides the architecture so that each partition implements some subset of the total instructions. This has the advantage that control only has to be passed from one partition to another between instruction executions, and it has the potential for process level concurrency between the partitions. An example of this is a machine with a separate floating point unit. Intra-chip partitioning generally allows for operator level parallelism between the partitions. This type of partitioning may enhance performance, and may make the machine easier to design and layout. APARTY is an automatic architectural partitioner that has the flexibility to produce both of these styles of partitions.

Partitioning results can be used to provide preliminary hardware information for CSTEP to use in making the control step schedule. CSTEP assumes that operators that are in separate partitions will not share hardware, so the partitioning information provides a lower bound on the amount of hardware to be used in creating the control step schedule.

Both the control step schedule and the partitioning information are used as inputs to EMUCS. EMUCS is a general algorithmic data path synthesis tool. It chooses when and how to bind operators and data carriers by attempting to minimize an overall cost for the design. The cost for each element is dependent on the amount of hardware needed and the interconnect costs; these costs are table driven. Partitioning information can be used by EMUCS to improve the design because partitions provide global insight into the design. The partitioning information suggests that operators in different partitions should be bound to different hardware, thus suggesting a high-level structure for the design and limiting the design space that EMUCS must search.

Busser accepts a complete data path description as input and attempts to improve the design by choosing appropriate busses. Busser uses a clique partitioning algorithm [Tseng84] to group module inputs together into busses so that there are no conflicts and the overall number of busses is minimized. Busses are chosen separately for each partition, and additional busses are chosen to interface the partitions.

The Architectural Partitioning Algorithm

The architectural partitioner APARTY is based on a standard clustering algorithm that operates by grouping objects together hierarchically according to some measure of closeness. This technique has been used in the past for design synthesis. McFarland [McFarland83] used clustering to partition a behavioral description, and used the partitioning information to guide the design of the data path. Rajan [Rajan85] and Kowalski's DAA [Kowalski84] both used clustering to obtain a preliminary indication of appropriate functional units. They used a single cluster stage that considered three criteria: data similarities, similarity of operators, and potential for operator level parallelism. McFarland's BUD [McFarland86] expanded this clustering to apply to all levels of partitioning, and used the clustering information to provide bottom-up design information. APARTY achieves additional flexibility over a basic clustering algorithm by allowing multiple clustering stages. The stages are applied consecutively, and each stage uses the results from the previous stage. Thus, the first stage clusters individual data flow operators into operator groups, and the next stage clusters those operator groups into groups to be used by the next stage. Each stage may have different closeness criteria that are used to determine how the operators are to be grouped. Four stages have been defined: control flow clustering, data flow clustering, inter-vtbody clustering, and common vtbody clustering.

The standard configuration of the stages begins with control clustering, which, for an instruction set architecture, may group the operators into instructions. Data flow clustering operates on these instructions, grouping them so that instructions that use common data carriers (e.g. floating point instruction making common use of floating point registers) are grouped together. Inter-vtbody clustering is run next to cluster instruction groups with their auxiliary vtbodies. For example, a vtbody that describes the floating point addition would be grouped with the group of floating point instructions. Finally, common vtbody clustering clusters instruction groups that use common auxiliary vtbodies. Thus, for an instruction set architecture, the standard configuration will produce an instruction set partition. To produce a different style of partition, the standard configuration can be modified by rearranging the order of the stages or by using only some of the stages.

Partitioning Results

APARTY has been used to partition several architectures into different styles of partitions. The IBM System/370 has been partitioned into an instruction set partition. APARTY has partitioned a DSP machine into address and data parts. Likewise, the 6502 common design was partitioned into address and data parts. A digital filter [Paulin87] was also partitioned, and the results were used to guide the later synthesis stages.

The digital filter that was partitioned has 34 add and multiply operators. Using the data clustering stage, the filter was partitioned into 2 parts, each containing both add and multiply operators. This implies that CSTEP can assume a minimum of 2 adders and 2 multipliers in making the control step schedule. Other assumptions in creating the schedule were that an add operation takes 1 cycle while a multiply requires 2 cycles. By limiting the hardware to its minimum for the partitions and allowing an operator to be scheduled only when hardware is available in its partition, CSTEP produces a 19 control step schedule for the filter. The shortest possible schedule for the 2 adder – 2 multiplier limit is at least one control step better than CSTEP finds, but this is a limitation of CSTEP's lookahead abilities rather than a limitation imposed by the partitions.

The resultant schedule was passed on to EMUCS, which produced a partitioned data path. This result was passed on to Busser which produced the final data path shown in Figure 4. In the figure, circles represent functional units, shaded rectangles are busses, unshaded rectangles are registers, trapezoids are muxes, and "C" represents a constant. The figure shows that each partition has 1 adder, 1 multiplier, and 3 busses which are input and output busses for the adder. A seventh bus connects the two partitions.

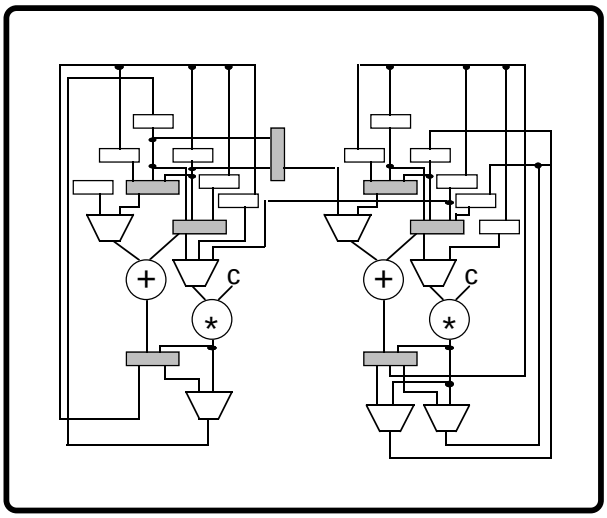

Figure 4
APARTY / EMUCS / Busser Filter Data Path

The information provided by the partitioner causes EMUCS to produce a design with 14% fewer muxes than it produces for the same design without making use of partitioning information. This improvement in the design can be attributed to the partitioner's ability to extract a high level view of the structure of the design, while EMUCS make design decisions based on an evaluation of a small window of the design. Thus the general path is a top-down approach to design, with the behavioral transformations allowing the designer to explore system level alternatives, the partitioner providing high level structural information, and the other tools filling in the specifics of the design.

SUGAR

SUGAR, the other synthesis path in the Workbench, is a style-specific approach to the automatic synthesis of microprocessors. In contrast to many design tools which apply the same general methods no matter what the design style, the approach used by SUGAR is based on the view that synthesis is a knowledge intensive problem requiring a large number of techniques specially tailored to individual design styles.

SUGAR uses a combination of algorithms and rules to encode knowledge about microprocessors. The knowledge about subsystems frequently found in commercial microprocessors is used to synthesize hardware for the behavior. For example, SUGAR recognizes subsystems such as instruction decode unit, condition code logic, branch logic, etc. SUGAR also has knowledge about what kind of bussing structures are frequently found in commercial microprocessors. This knowledge is used to design the global interconnections top-down.

Since microprocessor synthesis is a complex problem, SUGAR divides it into several smaller subproblems called phases. Each phase does a well-defined activity. The phases communicate with each other through a tree representation, rather than operating directly on the VT. Interactions between phases are reduced by delaying decisions until they can be made in an informed way. The phases do the following: (1) transform behavior to remove behavioral description inefficiencies, (2) restructure the control flow to allow fast decoding of instructions, (3) do classical compiler type flow analysis, (4) select an efficient bus structure, (5) allocate symbolic registers, (6) select micromachine code to implement the behavior, (7) schedule the code by assigning control steps to the register transfers, (8) improve the design by applying cost/speed tradeoffs, and (8) assign physical registers to the symbolic registers.

SUGAR Results

The fifth column in Table 1 shows some instruction execution times for the 6502. The unpipelined SUGAR design is slower than the common design example because that example combines the instruction fetch and decoding into one control step, whereas SUGAR and the other designs do not. The SUGAR design is also slower than the commercial design, due to its lack of an instruction prefetch, and due to a reduced number of data paths. Subject to these limitations, SUGAR produces control sequences that appear to be comparable to sequences produced manually.

Figure 5 shows the 6502 data path produced by SUGAR. This design was produced in approximately 2 hours of CPU time on a DEC VAXstation II.

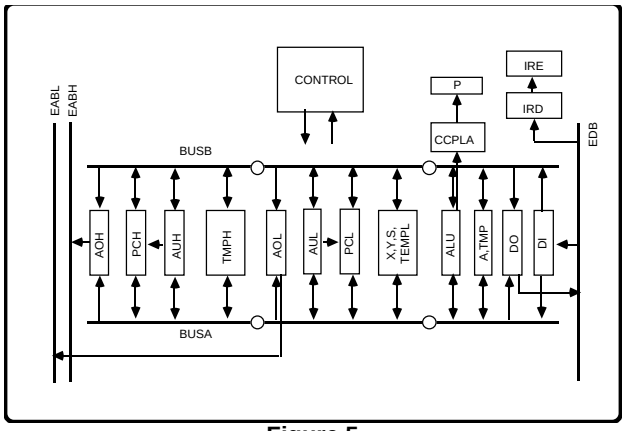

Figure 5
SUGAR 6502 Data Path

Conclusions

This paper has given an overview of the System Architect's Workbench, and has described two approaches to design in that system. One approach, using general design algorithms appropriate for designs of many styles, was demonstrated on various instruction set processors (the MCS6502 and the Berkeley Risc-1) and on an application specific design (the digital filter), and produced results comparable to commercial designs. The other approach, tuned to microprocessor design, was demonstrated on the MCS6502 and gives good results for that specific design style.

The 6502 common design example has not been synthesized by EMUCS because it would take in excess of 30 CPU hours on a DEC VAXstation II. However, results from the other stages of the general design path are available for this design. The behavioral transformations allow different high level design alternatives to be explored, producing the 6502 common design example, as well as a 2-stage 6502, both from a single "generic" starting description. The CSTEP control schedule for the unpipelined common design example can not be directly compared to that of the pipelined commercial design, but appears to be an acceptable schedule.

The digital filter was partitioned by APARTY, which does a good job of partitioning to reduce the overall interconnect. The results were then passed on to CSTEP and the data path synthesis programs to complete the synthesis path. A reasonable data path was produced in less than 4 minutes of CPU time, illustrating the broad applicability of the general approach to synthesis.

Compared to the general approach, the style-specific approach appears to produce faster synthesis programs. In contrast to EMUCS, SUGAR completed the design in 2 CPU hours. This difference in processing time is due to SUGAR's extensive knowledge of the microprocessor design style. For the ORA instruction, SUGAR produces control sequences comparable to those produced manually. Preliminary results for the Motorola MC68000 are equally promising.

Acknowledgements

This research was funded by the SRC under contract 86-01-068, by a grant from the IBM Corporation, and by the CMU CAD Industrial Affiliates. We would also like to thank R. Cloutier, D. Geiger, P. Koenig, D. Springer, R. Zimmermann, O. Amidi, K. Vissers and M. Doreau for their contributions.

References

[Blackburn 88] R.L. Blackburn, D.E. Thomas, P.M. Koenig. CORAL II: Linking Behavior and Structure in an IC Design System. In *Proc. of the 25th DAC*. ACM/IEEE, Anaheim, CA, June, 1988.

[Davidson 81] S. Davidson, D. Landskov, B.D. Shiriver and P.W. Mallett. Some Experiments in Local Microcode Compaction for Horizontal Machines. *IEEE Trans. on Computers* C-30(7):460-477, July, 1981.

[Fitzpatrick 81] D.T. Fitzpatrick, J.K. Foderaro, M.G.H. Katevenis, H.A. Landman, D.A. Patterson, J.B. Peek, Z. Peshkess, C.H. Sequin, R.W. Sherburne, K.S. Van Dyke. *A RISCy Approach to VLSI*. VLSI Design :14-20, Fourth Quarter, 1981.

[Girczyc 84] E.F. Girczyc and J.P. Knight. An Ada to Standard Cell Hardware Compiler Based on Graph Grammars and Scheduling. In *Proc. of ICCD-84*, pp. 726-731. IEEE, Port Chester, NY, Oct., 1984.

[Hitchcock 83] C.Y. Hitchcock III and D.E. Thomas. A Method of Automatic Data Path Synthesis. In *Proc. of the 20th DAC*, pp. 484-489. ACM/IEEE, Miami, FL, June, 1983.

[Kowalski 84] T.J. Kowalski. *The VLSI Design Automation Assistant: A Knowledge-Based Expert System*. PhD thesis, ECE Dept., Carnegie-Mellon University, April, 1984. SRC Report CMUCAD-84-29.

[McFarland 83] M.C. McFarland, S.J. Computer-Aided Partitioning of Behavioral Hardware Descriptions. In *Proc. of the 20th DAC*, pp. 472-478. ACM/IEEE, Miami, FL, June, 1983.

[McFarland 86] M.C. McFarland, S.J. Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions. In *Proc. of the 23rd DAC*, pp. 474-480. ACM/IEEE, Las Vegas, NV, June, 1986.

[Nestor 87] J.A. Nestor. *Specification and Synthesis of Digital Systems with Interfaces*. PhD thesis, ECE Dept., Carnegie-Mellon University, April, 1987. SRC Report CMUCAD-87-10.

[Pangrle87] B.M. Pangrle and D.D. Gajski. Design Tools for Intelligent Silicon Compilation. *Transactions on CAD* CAD-6(6):1098-1112, November, 1987.

[Parker 86] A.C. Parker, J. Pizarro, and M. Mlinar. MAHA: A Program for Datapath Synthesis. In *Proc. of the 23rd DAC*, pp. 461-466. ACM/IEEE, Las Vegas, NV, June, 1986.

[Paulin 87] P.G. Paulin and J.P. Knight. Force-Directed Scheduling in Automatic Data Path Synthesis. In *Proc. of the 24th DAC*, pp. 195-202. ACM/IEEE, Miami, FL, June, 1987.

[Rajan 85] J.V. Rajan and D.E. Thomas. Synthesis by Delayed Binding of Decisions. In *Proc. of the 22nd DAC*, pp. 367-373. ACM/IEEE, Las Vegas, NV, June, 1985.

[Snow 78] E.A. Snow. *Automation of Module Set Independent Register-Transfer Level Design*. PhD thesis, EE Dept., Carnegie-Mellon University, April, 1978.

[Thomas 81] D.E. Thomas and D.P. Siewiorek. Measuring Design Performance to Verify Design Automation Systems. *IEEE Trans. on Computers* C-30, January, 1981.

[Trickey 87] H. Trickey. Flamel: A High-Level Hardware Compiler. *Transactions on CAD* CAD-6(2):259-269, March, 1987.

[Tseng 83] C.J. Tseng and D.P. Siewiorek. Facet: A Procedure for the Automated Synthesis of Digital Systems. In *Proc. of the 20th DAC,* pp. 490-496. ACM/IEEE, Miami, FL, June, 1983.

[Vasantharajan 82] J. Vasantharajan. *Design and Implementation of a VT-Based Multi-Level Representation*. Master's thesis, EE Dept., Carrnegie-Mellon University, February, 1982.

[Walker 87] R.A. Walker and D.E. Thomas. Design Representation and Transformation in The System Architect's Workbench. In *Proc. of ICCAD-87*, pp. 166-169. IEEE, Santa Clara, CA, November, 1987.