Behavioral Transformation for Algorithmic Level IC Design

ROBERT A. WALKER, MEMBER, IEEE, AND DONALD E. THOMAS, FELLOW, IEEE

Abstract – Now that the field of automated synthesis for register transfer level integrated circuit design is beginning to mature, it is appropriate to begin developing tools for higher levels of design. At the next higher level, it is appropriate to explore behavioral and structural partitioning, answering such questions about the design as:

- Should the design be implemented on a single VLSI chip, or partitioned into two or more chips, and if it is to be partitioned, where should the behavior be divided?
- Should the design be implemented as a single process, with a single data path and controller, or should it be split into two or more processes, each hopefully smaller and faster than the single process design and with more potential concurrency?
- Should the design be pipelined or left unpipelined, and if pipelined, how many stage divisions should there be, and where should they be placed?

The goal of this research was to define the Algorithmic Level of design (also known as the Behavioral Level) and to provide the designer with the means to explore these issues. Within the framework of the System Architect's Workbench, a new set of behavioral and structural transformations was developed to allow the interactive exploration of Algorithmic Level design alternatives. This paper describes these transformations and presents a set of examples, both to demonstrate the application of the transformations, and to further illustrate their effects.[†]

I. INTRODUCTION

Over the past few years, Computer Aided Design (CAD) and design automation (DA) have begun to gain acceptance at the Functional Block (Register Transfer) Level[1] of abstraction. Here the emphasis of CAD and DA tools is on synthesizing a design in terms of ALUs, registers, multiplexors, and a control sequence table, given a behavioral description of the hardware to be designed. An excellent tutorial to ongoing work at this level is [5].

As the field of Functional Block Level design automation matures, it is appropriate to begin developing tools for higher levels of design. At the next higher level, the Algorithmic Level (also known as the Behavioral Level), it is appropriate to explore behavioral and structural partitioning. It is at this level that the designer should answer such questions about the design as:

- Should the design be implemented on a single VLSI chip, or partitioned into two or more chips, and if it is to be partitioned, where should the behavior be divided?
- Should the design be implemented as a single process, with a single data path and controller, or should it be split into two or more processes, each hopefully smaller and faster than the single process design and with more potential concurrency?
- Should the design be pipelined or left unpipelined, and if pipelined, how many stage divisions should there be, and where should they be placed?

Research into these questions is just beginning. The Flamel system [12] automatically applies loop unrolling and other basic-block transformations to improve the parallelism of a design. The Yorktown Silicon Compiler [3] partitions its designs, although its partitioning is performed at the Functional Block Level, and is used primarily to limit the size of the design being synthesized. The Sehwa system [7] automatically determines stage divisions and pipelines designs into a fixed number of stages with the maximum performance, and the HAL system [8] pipelines designs (with user-specified stage boundaries) so as to balance the distribution of resources across groups of concurrent control steps.

The goal of the research described in this paper, as well as related research at Carnegie Mellon, is to provide the designer with the means to explore these questions in an interactive fashion. Toward this goal, we have developed a set of *behavioral and structural transformations* [15, 16] to allow the exploration of Algorithmic Level design alternatives. Thus the designer can start with an initial description of a design, interactively transform its behavior or structure, examine the results in terms of the control schedule or synthesized data path, and iterate to explore other design alternatives. Note that our current implementation provides an interactive framework for the designer to manually explore design alternatives, not an automated system that attempts to produce a single "best" design.

The development of this set of transformations was carried out in the framework of the System Architect's Workbench [11, 15], a DA tool aimed at the automatic synthesis of digital systems. The inputs to the Workbench are a behavioral description of a piece of hardware to be designed, written in the ISPS language [1], and a set of interface constraints. This behavioral description is mapped onto a directed acyclic graph called the Value Trace [9], or VT, which represents the design as a set of dataflow operations. These operations are then

[†] Manuscript received June 23, 1988; revised February 7, 1989 and April 10, 1989. This work was supported by the Semiconductor Research Corporation. The review of this paper was arranged by Associate Editor M.R. Lightner.

R.A. Walker is with the Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY 12180.

D.E. Thomas is with the Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, 15213.

IEEE Log Number 8929567.

[1] These levels are defined in more detail in [14].

synthesized as an abstract set of Functional Block Level data path components and a control sequence table, and eventually as standard cells and a microcoded or PLA-based controller.

Some of these transformations, together with transformations developed earlier as part of the CMU-DA system [10], are based on compiler transformations, and are used to transform the behavioral representation from one based on a readable input description to one with a more efficient control structure:

- Procedures can be expanded inline to eliminate unnecessary microsubroutine jumps.
- Individual operations can be moved into and out of the branches of decoding operations, often achieving a better packing of operations into control steps.
- Nested decoding operations can be combined, eliminating unnecessary levels of decoding.

Other transformations have a more extensive effect, dramatically effecting the implementation of the design:

- Parts of the behavioral description can be transformed into concurrent processes, signifying that they are to be synthesized with separate controllers.
- The behavioral description can be pipelined, with the designer specifying both the number of stages and the placement of stage boundaries.
- The behavioral description can be structurally partitioned, defining chip or board boundaries.

Using both sets of transformations, a designer can begin with a single "generic" behavioral description, transform the design to obtain a more efficient control structure, and behaviorally or structurally transform that design to explore further Algorithmic Level design alternatives, such as concurrency or pipelining. As currently implemented in the System Architect's Workbench, these transformations can be applied to the VT interactively, the results can be analyzed (for example, by comparing the number of control steps or the amount of communication overhead), and additional transformations applied as necessary. The ability to consider alternative Algorithmic Level organizations in a CAD environment based on a single behavioral description is a unique capability of the Workbench.

This paper describes these transformations, concentrating primarily on the effect of each transformation on the VT, rather than on the synthesized data path and controller. Since the operations in the VT do not map one-to-one onto components in the synthesized design, the effect of each transformation on the data path and controller is often difficult to observe. Additionally, many of the newer synthesis tools attempt to consider the entire behavioral representation as a unit, rather than as linked basic blocks, making it difficult to determine the effect of a (relatively) local transformation on the entire control step schedule and data path. Due to these concerns, this paper concentrates on the effect of each transformation on the VT, and presents only brief insights into its effect on the synthesized design.

Depending on the results desired and the design alternatives being explored, each transformation may be applied in a variety of ways. For example, the SELECT Motion transformations of Section III.A may be used to facilitate other transformations, to obtain a more efficient control structure, or as the first step in forming stages for a pipelined design. The pipelining transformations may be used to form a functionally-decomposed pipeline, or to form one with "balanced" stages. There is no single "right" way to use each transformation.

The next two sections of this paper, Sections II and III, present transformations that can be used to transform a design from one based on a readable description to one with a more efficient control structure. Section IV defines a transformation for creating processes, and Section VI defines a transformation for creating pipestages. Each of these sections defines the effect of the transformation on the VT, describes how the transformation can be used, and briefly presents some insights into its effect on the synthesized design.

Two of the remaining sections illustrate the use and effects of the transformations. Section V shows how the transformations can be used interactively to split a digital filter VT into two processes, and Section VII illustrates the use of other transformations to interactively pipeline an unpipelined MOS Technology MCS6502 VT. Finally, Section VIII summarizes the paper and presents some conclusions.

II. VTBODY TRANSFORMATIONS

When the ISPS behavioral description is mapped onto the VT, procedures and labeled blocks of operations are mapped onto VT subgraphs called *vtbodies*. CALL and LEAVE operations are used to transfer the flow of control between these vtbodies, similar to the CALL and RETURN operations in many programming languages.

The vtbody transformations [9] allow the designer to remove and add CALLs between vtbodies by expanding vtbodies inline and forming new vtbodies, respectively. Since each CALL in the VT is implemented by a microsubroutine jump in the final design (assuming a microprogrammed controller), these transformations give the designer control over the placement of the microsubroutine jumps. Although these transformations were not defined as part of the authors' recent research, they are included here because they are part of the Workbench and are used in later examples.

A. Vtbody Inline Expansion

As defined by Snow, *Vtbody Inline Expansion* replaces a CALL operator with the contents of the called vtbody. Three versions of this transformation are implemented in the Workbench: one expands inline the vtbodies specified by the user, one expands inline all vtbodies that are called only once, and a third recursively expands inline all vtbodies beginning at a vtbody specified by the user. In Fig. 1, vtbody V2:B is expanded inline, resulting in the VT shown on the right side of the figure.

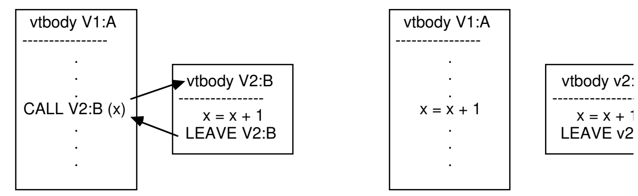

Fig. 1. Before and After Vtbody Inline Expansion

In designs synthesized by the Workbench, a CALL operation performs a microsubroutine jump, and a LEAVE operation performs a microsubroutine return. Thus, if the VT in Fig. 1 was synthesized before the transformation, at least three cycles would be required for its execution: one to execute the CALL and the instructions before it, one to execute the operations in the called vtbody, and one to execute the operations following the CALL. In contrast, after the transformation it is possible that all the remaining operations could be packed into a single cycle, depending on the execution time of each operation and the length of the cycle.

B. Dead Vtbody Elimination

As defined by Snow, *Dead Vtbody Elimination* removes a vtbody that is not called by other vtbodies. Two versions of this transformation are implemented in the Workbench: one removes the dead vtbody specified by the user, and the other removes all dead vtbodies in the VT. For example, applying Dead Vtbody Elimination to the VT on the right side of Fig. 1 would remove vtbody V2:B, which is no longer referenced by other vtbodies. This transformation is typically applied after Vtbody Inline Expansion.

C. Vtbody Formation

Vtbody Formation, as defined by Snow, encapsulates a specified set of operators into a new vtbody, which is then CALLed from the original vtbody. This transformation is the inverse of Vtbody Inline Expansion.

III. SELECT TRANSFORMATIONS

In ISPS, IF operations are used for conditional branching, and DECODE operations are used for CASE-like decoding. Both of these operations are mapped onto the VT *SELECT* operation. This SELECT operation decodes a value (called a *selector*), and based on the decoded value, chooses a single branch to execute from a set of alternative branches. Associated with each branch is a set of *activation values* that must contain the selector for the branch to be chosen. At any given time, only a single branch of the SELECT can be active.

The SELECT transformations allow the designer to interactively manipulate SELECTs and the operators surrounding them. Operators can be moved into and out of SELECTs, nested SELECTs can be combined, and SELECTs can be duplicated.

All moves are subject to data dependencies, in that an operator can not be moved before an operator on which it is data dependent. Given an operator O_x, with the set of outputs O_xO, and operator O_y, with the set of inputs O_yI, O_y is said to be *data dependent* on O_x if:

$$\exists\, V \in O_xO,\ V \in O_yI$$

Since data dependency is transitive, operator O_y can also be data dependent on operator O_x if:

$\exists\, O_z$,
O_y is data dependent on O_z, and
O_z is data dependent on O_x

In this section, the following notation will be used. Given SELECT S_a, the ordered set of m branches of SELECT S_a will be represented by $S_aB = < S_aB_1, \ldots, S_aB_m >$, and the ordered set of n operators contained in branch S_aB_j of SELECT S_a will be represented by $S_aB_jO = < S_aB_jO_1, \ldots, S_aB_jO_n >$. The symbol “&” will be used to indicate concatenation of two ordered sets.

In the simplified VT diagrams in this section (Figs. 2-5), VT operators are represented by circles and SELECTS are represented by squares. SELECT branches and their contents are represented by five-sided polygons and the operations below them. For example, Fig. 2 illustrates a SELECT with three branches, each branch containing two operators, and an operator before and after the SELECT.

A. SELECT Motion

The SELECT Motion transformations, some of which were defined earlier in [9] or [13], are applied interactively to move operators into or out of SELECTs as shown in Fig. 2[2]. SELECT motion can be used to reform the control structure, to achieve a better control step packing, or to facilitate other transformations.

SELECT Motion Down Into SELECT moves an operator O_a from above a SELECT S_a into the top of all the SELECT’s branches S_aB. Thus if S_aB_jO' designates the (new) ordered set of operators in branch S_aB_j after the transformation:

Before SELECT Motion Down Into SELECT:
Given an ordered set of operators $O = < O_a, S_a >$
After SELECT Motion Down Into SELECT:
$O = < S_a >$,
$\forall\, S_aB_j \in S_aB$,
$S_aB_jO' = < O_a > \& S_aB_jO$

SELECT Motion Up Into SELECT is defined similarly, moving an operator O_b from below a SELECT S_a into the bottom of all the SELECT’s branches S_aB.

Other SELECT Motion transformations move operations out of the SELECTs. *SELECT Motion Down Out of SELECT*

[2] SELECT Motion was originally defined by Snow for data SELECTs, which were essentially data multiplexors. However, when the VT was implemented, those data SELECTs were abandoned in favor of control SELECTs, which choose a single branch to execute from a set of alternative branches. SELECT Motion Up Into SELECT and Motion Down Out of SELECT were defined for the present form of SELECTs in [13]. The work described in this paper completes the set of four SELECT Motion transformation.

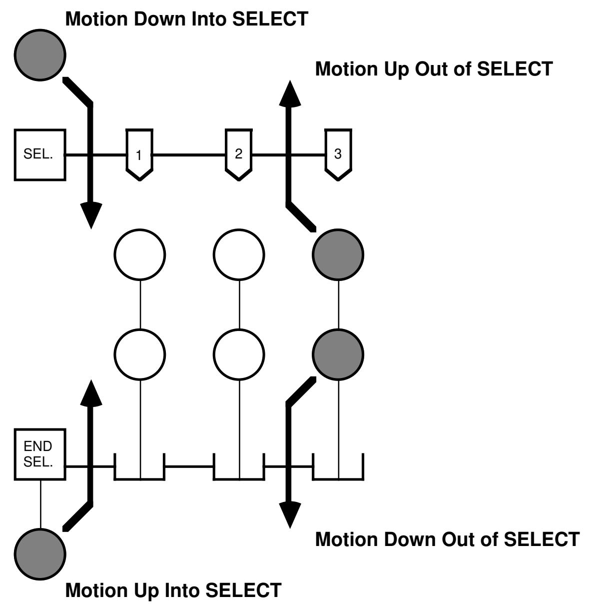

Fig. 2. SELECT Motion Alternatives

moves an operator O_d from inside a branch of SELECT S_a to below the SELECT, and deletes duplicates of that operator from the other branches:

Before SELECT Motion Down Out of SELECT:

Given an ordered set of operators $O = < S_a >$,

$\forall\ S_aB_j \in S_aB$,

$S_aB_jO' = S_aB_jO\ \&\ < O_d >$

After SELECT Motion Down Out of SELECT:

$O = < S_a, O_d >$,

$\forall\ S_aB_j \in S_aB$,

$S_aB_jO'' = S_aB_jO$

SELECT Motion Up Out of SELECT is defined similarly, but moves an operator O_u from inside a branch of SELECT S_a to above the SELECT, and deletes duplicates of that operator in the other branches.

SELECT Motion can be applied for a variety of reasons. It can be used to facilitate other transformations, particularly other SELECT transformations (e.g., SELECT Combination), as described in the example of Section VII. It can be used to obtain a better packing of operators into control steps, although with some control step schedulers this effect is difficult to predict, and the global nature of other control step schedulers may render this motion unnecessary. Finally, SELECT Motion may be used, together with other SELECT transformations, to form the SELECT “stages” that are necessary as the first step in the Pipestage Creation transformation of Section VI.

B. SELECT Combination

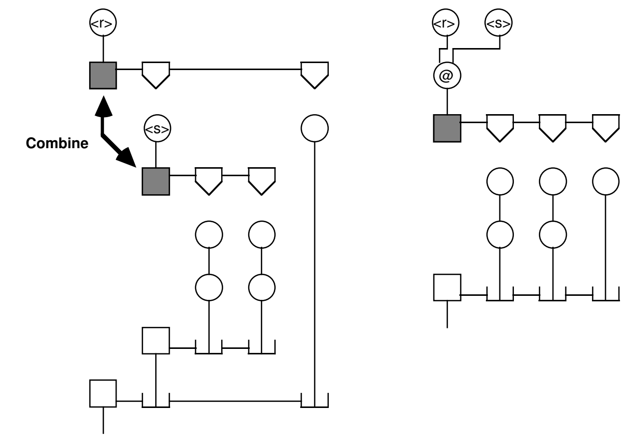

Fig. 3. Before and After SELECT Combination

The *SELECT Combination* transformation[3] is applied interactively to combine two nested SELECTs into one, as shown in Fig. 3. The branch of the outer SELECT that contains the inner SELECT is replaced with all the branches of the inner SELECT, the branch activation values are updated accordingly, and the inner SELECT is removed. Thus, SELECT Combination replaces two sequential decoding operations with a single decoding operation.

SELECT Combination reforms the branch structure as follows, moving the branches of the inner SELECT S_i into the outer SELECT S_o:

Before SELECT Combination:

Given a set of SELECTs $S = \{ S_i, S_o \}$,

$O = < O_x, S_o >$,

$S_oB =$

$< S_oB_1, \ldots, S_oB_i, S_oB_j, S_oB_k, \ldots, S_oB_m >$,

$S_oB_jO = < O_y, S_i >$

After SELECT Combination:

$S = \{ S_o \}$,

$O = < O_x, O_y, S_o >$,

$S_oB =$

$< S_oB_1, \ldots, S_oB_i >\ \&\ S_iB\ \&$

$< S_oB_k, \ldots, S_oB_m >$

Two versions of SELECT Combination have been defined. In the first version, the selectors of the two SELECTs are concatenated to form a new selector, as shown in Fig. 3. In this figure, “<r>” indicates reading a bit field of value “r”, and “@” represents concatenation of two values. In the second version, if the two selectors reference either the same value or subfields read from the same value, the full value is used as the resulting selector. In either case, the branch activation values are updated as appropriate for the new selector.

SELECT Combination is typically applied to combine two or more sequential decoding operations into a single operation. For example, many behavioral descriptions of

[3] SELECT Combination was also defined for data SELECTs by Snow, although data SELECTS were never actually implemented in the VT. The work described in this paper defines SELECT Combination for the present form of SELECTs.

processors are written with several levels of decoding — one decoding operation to decode the instruction's "group" (arithmetic, data transfer, etc.), another to decode the opcode, and another to decode the addressing mode. Using SELECT Combination two or more of these decoding operations can be combined; this effect is demonstrated in the example of Section VII. SELECT Combination usually results in a faster design, although possibly increasing the complexity of the controller by forcing it to evaluate more complex expressions.

C. SELECT Duplication

The *SELECT Duplication* transformation is applied interactively to duplicate a SELECT, not including the operators in its branches, as shown in Fig. 4. When a SELECT is duplicated one or more times in this manner, it is assumed that the first SELECT in the series will perform the decoding operation once, and pass microcode dispatch addresses to the following SELECTs. This transformation could be used in preparation for functional branching or pipelining.

SELECT Duplication duplicates a SELECT S_a, producing a new SELECT S_d with empty branches as follows:

Before SELECT Duplication:

Given an ordered set of operators $O = < S_a >$

After SELECT Duplication:

$O = < S_d, S_a >$,

$\forall\, S_dB_j \in S_dB$,

$S_dB_jO = < >$

SELECT Duplication is typically applied when creating a pipelined design. First, a set of "stages" are formed using SELECTs, and then these stages are encapsulated into vtbodies and transformed into pipestages. To form the SELECT stages, SELECT Duplication is used to duplicate the instruction decoding SELECT for all but the last stage, and SELECT-to-SELECT Transfer (defined in the next section) is used to move operators into the appropriate stage.

D. SELECT-to-SELECT Transfer

The *SELECT-to-SELECT Transfer* transformation is applied interactively to move an operator from a branch in one SELECT to the end of the corresponding branch in the immediately preceding duplicate SELECT, as shown in Fig. 5. Note that the output of the moved operator must now pass through the output of the first SELECT, to be multiplexed with outputs of other branches if necessary. If each SELECT in the figure will later become a separate pipestage, this transformation moves the shaded operator into an earlier stage.

SELECT-to-SELECT Transfer moves an operator O_a from a branch S_aB_a in SELECT S_a to the end of the corresponding branch in the preceding SELECT S_p, as follows:

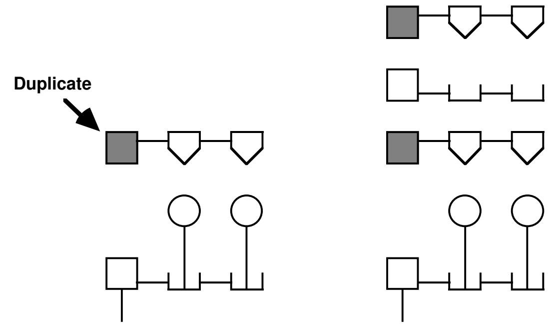

Fig. 4. Before and After SELECT Duplication

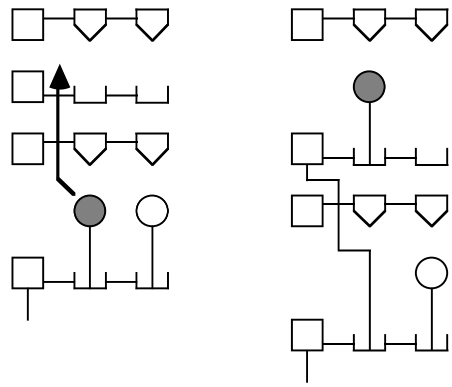

Fig. 5. Before and After SELECT-to-SELECT Transfer

Before SELECT -to-SELECT Transfer:

Given an ordered set of operators $O = < S_p, S_a >$,

$\exists\, S_aB_a \in S_aB$,

$S_aB_aO' = < O_a >$ & S_aB_aO,

$\exists\, S_pB_a \in S_pB$, S_pB_a corresponds to S_aB_a,

$S_pB_aO' = S_pB_aO$,

After SELECT -to-SELECT Transfer:

$O = < S_p, S_a >$,

$S_aB_aO'' = S_aB_aO$,

$S_pB_aO'' = S_pB_aO$ & $< O_a >$

SELECT-to-SELECT Transfer is typically used, together with Select Duplication, to form a set of "stages" as the first step in creating a pipelined design. In contrast to the Sehwa system [7], which automatically chooses the pipestage boundaries to produce "balanced" stages, these transformation let the designer interactively choose the stage boundaries. This choice gives the designer the flexibility to produce a wider range of design alternatives, from balanced stages similar to those produced by Sehwa, to the functionally-decomposed stages (for example, an instruction decoding stage, an operand fetch stage, and an execution stage) found in many large processors.

Future work may attempt to automate this process. Methods similar to those employed by Sehwa could be used to produce balanced stages, and context analysis could be used to aid the designer in the formation of functionally-decomposed stages.

IV. Process Creation

Previous sections of this paper have described a set of vtbody and SELECT transformations. Those transformations, some of which are similar to the transformations used by optimizing compilers, are often used to produce a more efficient control structure for the design. In contrast, the transformations described in the remainder of this paper have a more extensive effect, dramatically changing the implementation of the design. These transformations allow the designer to explore Algorithmic Level design alternatives: forming concurrent processes and pipelining the design's dataflow.

The first of these Algorithmic Level transformations, the *Process Creation* transformation, creates a process from the specified vtbody and its children. It marks the specified vtbody as a process, replaces each CALL to the vtbody with a SEND / RECV pair of message-passing operators, replaces each LEAVE from the vtbody with another SEND / RECV pair, and adds a RESTART operator to the vtbody so that it will execute continuously. The new process will execute concurrently with the other processes, and will be synthesized with a separate data path and controller.

An example of Process Creation is shown in Figs. 6 and 7. In this example, before the transformation, vtbody V1:A CALLs vtbody V2:B, passing it values x and y. After the transformation, the CALL has been replaced with a SEND operator, and a corresponding RECV operator has been added to the beginning of vtbody V2:B; values x and y are now passed via these message-passing operators. The LEAVE operator is replaced by a similar pair of SEND / RECV operators. After the transformation, vtbody V2:B is marked as a process, and is RESTARTed to execute continuously.

This transformation significantly affects the final design, since the Workbench synthesizes each process as a separate data path and controller. The resulting set of controllers is assumed to have a common clock, and the SEND and RECV operators are used both for synchronization and data transfer between the processes. At the present time, the Workbench does not generate synchronization hardware for the SEND and RECV operations, but can generate the proper control sequence for non-blocking NBSEND and NBRECV operations. We hope to eventually support the blocking SEND and RECV operations as well.

A designer might choose to apply Process Creation for a variety of reasons. If a design is too large to fit onto a single chip, Process Creation might be applied to split the design into two processes, each occupying less area than the original design. If a design is too slow, Process Creation might be used to introduce more concurrency, or simply to produce two processes, each with a smaller layout than the original design, and thus with reduced path lengths, a faster clock, and improved performance (assuming the delay due to interprocess communication is not too large).

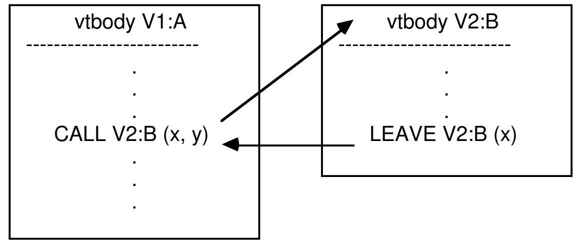

Fig. 6. Before Process Creation

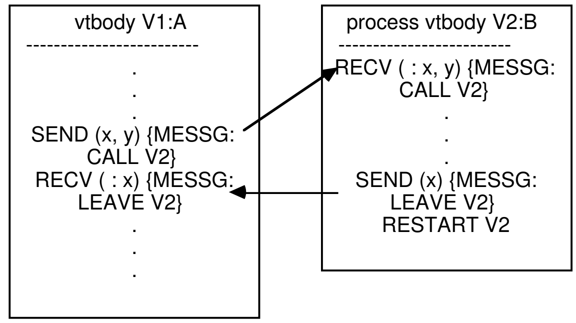

Fig. 7. After Process Creation

When Process Creation splits the VT into two processes, the Workbench synthesizes two data paths. Additional processors (ALUs, adders, etc.) are added if necessary, and additional registers and multiplexors are typically added due to the inter-process communication overhead. If the amount of additional hardware is small enough relative to the total amount of hardware, designs that could not fit onto a single chip when synthesized as one process can be split into two processes, each small enough to fit onto a chip of that size. The example in the next section demonstrates this reduction in per-process area.

Since Process Creation introduces a new controller, it affects both the control step schedule and the amount of microcode required to control the design. With the proper control step schedule, adding a second process can introduce additional concurrency into the design. Furthermore, since each of the two processes often has fewer states than the single-process design, the microword width in each process often decreases (fewer bits are needed to specify the next state), reducing the total amount of microcode for the design. The example in the next section also demonstrates this effect.

These two effects (reduction in per-process area and microcode) are not guaranteed to occur in all designs, or for all process divisions. However, they do occur for some designs, and some process divisions, as demonstrated by the examples in the next section. For further details, see [16].

V. An Example of Process Creation – A 5th Order Elliptic Wave Filter

This example, a fifth order digital elliptic wave filter from [4], was one of the examples from the ACM / IEEE 1988 Workshop on High-Level Synthesis [2]. The VT for the filter is shown in Fig. 8.

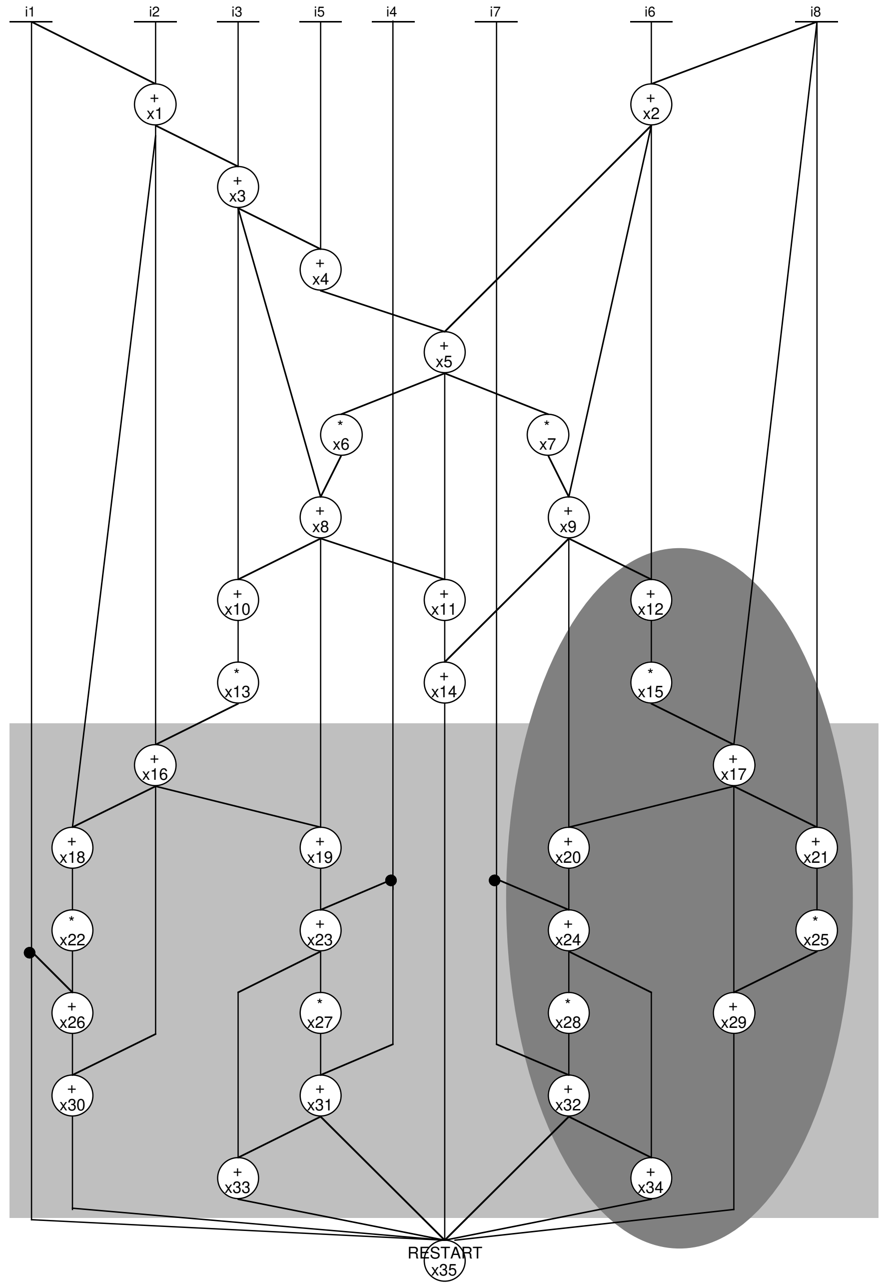

Fig. 8. Filter VT Before Process Creation

Section A describes the transformations required to split the VT horizontally, encapsulating the operators in the light gray rectangle in Fig. 8 into a separate process. Section B describes the transformations required to split the VT vertically, encapsulating the operators in the dark gray oval into a separate process. Synthesis results from the CSTEP control step scheduler [11, 6] and the EMUCS data path allocator [11] are presented for all designs.

A. A Two-Process Filter

This section describes the transformations required to interactively split the filter's VT horizontally into two processes, gives synthesis results for the single-process and two-process designs, and presents some insights into how the transformation process caused changes in the results. In Fig. 8, the operators in the light gray rectangle will become the second process; this division was chosen by the designer to split the VT horizontally into two processes with roughly the same number of operations and control steps.

The first step in transforming the design is to encapsulate the operators that are to become the second process into a new vtbody. If the second process is to contain operators x16 through x34 (the operators inside the light gray rectangle in Fig. 8), those operators are encapsulated into a new vtbody using the Vtbody Formation transformation.

TABLE I
EFFECT OF PROCESS CREATION ON FILTER'S MICROCODE

design	µwords (CSTEP)	bits per µword to specify next state (CSTEP)	bits per µword to specify µop (AM 2910)	bits of µcode for control	total change in µcode for control	bits per µword to control muxes (EMUCS)	bits per µword to control registers (EMUCS)	bits of µcode to control data path	total change in µcode to control data path	total µcode change
single process	17	5	4	153		29	11	680		
hor. split	1	1	4	5		0	8	8		
at x3	16	4	4	128	-13.1%	35	10	720	7.1%	3.4%
hor. split	10	4	4	80		14	10	240		
at x16	7	3	4	49	-15.7%	26	9	245	-28.7%	-26.3%
hor. split	14	4	4	112		30	10	560		
at x26	4	2	4	24	-11.1%	13	8	84	-5.3%	-6.4%
vertical	17	5	4	153		22	10	544		
split	10	4	4	80	52.3%	11	4	150	2.1%	11.3%

Once the new vtbody has been formed, it can be converted into a separate process using the Process Creation transformation. CALLs and LEAVEs to and from that vtbody are removed and replaced with SEND and RECV message-passing operations, and a RESTART operator is added to the process so that it will always be active.

The CSTEP control step scheduler and the EMUCS data path allocator synthesis results for the single-process design and a set of horizontally-split two-process designs are shown in Tables I and II. Table I shows the effect of Process Creation on the filter's microcode, and Table II shows the effect on the filter's data path. Both tables show the results for the single-process design, the results for three different horizontally-split two-process designs, and the change due to Process Creation. The bottom row of each table will be discussed later. These results assume that addition takes one cycle, and multiplication takes two cycles.

Table I summarizes the changes in the design's microcode as a result of Process Creation. For each design, the columns in the table are:

µwords (CSTEP) – for each process, the number of microwords (control steps) per process, based on CSTEP's control step schedule

bits per µword to specify the next state (CSTEP) – for each process, the number of bits required to specify the next state in each process, based on the total number of states produced by CSTEP

bits per µword to specify µop (Am2910) – for each process, the number of bits required to specify the controller's microoperation to be performed, based on an assumed Am2910 controller

bits of µcode for control – for each process, the total number of bits of microcode required for control

total change in µcode for control – for the entire design, the percent change in the total number of bits of microcode needed for control between the single-process design and the two-process design

bits per µword to control muxes (EMUCS), bits per µword to control registers (EMUCS) – for each process, the number of bits required to control the multiplexors and registers in the data path produced by EMUCS

bits of µcode to control data path – for each process, the total number of bits of microcode required to control the data path

total change in µcode to control data path– for the entire design, the percent change in the total number of bits of microcode needed to control the data path between the single-process design and the two-process design

total µcode change – the total percent change in the number of bits of microcode between the single-process design and the two-process design

For horizontal splits, the table shows several effects of Process Creation on the design's microcode. It shows, for the splits before x3 and x16, no change in the total number of microwords — each design still executes in 17 cycles (assuming 1 cycle per control step). In contrast, an extra cycle is required for the split before x26, because operations before and after x26 that were previously scheduled into the same control step were scheduled into two different control steps after the process division.For horizontal splits, the table also shows an 11 to 16 percent decrease in the total amount of microcode required for control, due primarily to a decrease in the number of states per process (meaning that fewer bits are required to specify the next state). In general, the largest decrease in the total amount of microcode required for control occurs when the two stages contain an equal number of control steps. In this example, the division closest to meeting this criterion is the horizontal split before x16, where the two processes contain 10 and 7 control steps, respectively; this split produces a 15.7% decrease in the total amount of microcode required for control. As the split is moved away from this division, a lesser decrease occurs.

For horizontal splits, the table also shows between a 7 percent increase and a 29 percent decrease in the total amount of microcode required to control the data path. This effect depends on the changes to the data path produced by EMUCS; in this example, the number of registers and multiplexors per process generally decreases, so the amount of microcode required to control those components decreases as well. Furthermore, the total amount of microcode required to control the data path may decrease even though the amount of hardware in both data paths increases. For example, before

TABLE II
EFFECT OF PROCESS CREATION ON FILTER'S DATA PATH

design	values xferred between processes	16-bit add procs. (EMUCS)	area (sq. mm) (MOSIS 3µm CMOS)	16-bit mult. procs. (EMUCS)	area (sq. mm) (MOSIS 3µm CMOS)	16-bit registers (EMUCS)	area (sq. mm) (MOSIS 3µm CMOS)	mux input bits (EMUCS)	area (sq. mm) (MOSIS 3µm CMOS)	total area (sq. mm)	ratio of area to single process design
single process		4	4.0	4	1017.0	11	1.0	848	14.8	1036.7	
hor. split	3	2	2.0	0	0.0	8	0.7	0	0.0	2.7	0.3%
at x3	2	4	4.0	4	1017.0	10	0.9	1024	17.8	1039.7	100.3%
hor. split	6	3	3.0	2	508.5	10	0.9	400	7.0	519.3	50.1%
at x16	3	4	4.0	4	1017.0	9	0.8	704	12.3	1034.0	99.7%
hor. split	9	4	4.0	2	508.5	10	0.9	832	14.5	527.8	50.9%
at x26	6	2	2.0	2	508.5	8	0.7	384	6.7	517.9	50.0%
vertical	3	2	2.0	2	508.5	10	0.9	608	10.6	521.9	50.3%
split	2	2	2.0	2	508.5	4	0.3	304	5.3	516.1	49.8%

Process Creation, the filter design has 17 microwords, requiring a total of (17) (29 + 11) = 680 bits to control the data path. For the horizontal split before x16, the first process has 10 microwords, requiring only (10) (14 + 10) = 240 bits to control the data path, and the second process has 7 microwords, requiring only (7) (26 + 9) = 245 bits to control the data path, for a total of 485 bits. Even though the total amount of hardware has increased (see Table II), the amount of microcode decreases due to the decrease in the number of data path components that have to be controlled in each process.

Table II summarizes the changes in the design's data path as a result of Process Creation. All area estimates are based on MOSIS 3µm CMOS cells; no routing area is included other than that inside each cell. For each design, the columns in the table are:

values xferred between processes – the number of (16-bit) values transferred between the first process and the second, and between the second process and the first

16-bit add procs. (EMUCS) – for each process, the number of 16-bit add processors in each data path produced by EMUCS

area – for each process, the estimated area required for the add processors; based on four 4-bit adders per 16-bit adder, with each 4-bit adder occupying 363µm x 684µm

16-bit mult. procs. (EMUCS) – for each process, the number of 16-bit multiplication processors in each data path produced by EMUCS

area – for each process, the estimated area required for the multiplication processors; based on a 16x16 array of add processors

16-bit registers (EMUCS) – for each process, the number of 16-bit registers in each data path produced by EMUCS

area – for each process, the estimated area required for the registers; based on 16 single-bit non-inverting buffers per 16-bit register, with each buffer occupying 150µm x 36µm

mux input bits (EMUCS) – for each process, the number of multiplexor input bits in each data path produced by EMUCS

area – for each process, the estimated area required for the muxes; an approximation based on an 8-input, single-bit mux occupying 363µm x 384µm

total area – for each process, the total estimated area

ratio of area to single process design – for each process, the ratio of the area of that process to the area of the single process design

For horizontal splits, the table shows several effects of Process Creation on the data path. As explained in the introduction, since EMUCS takes a global view of the design, it is difficult to predict these effects in advance. In general, the table shows that the total number of processors, registers, and mux input bits increases for a two-process design, yet the number in each process is often smaller than the single-process design.An important effect of Process Creation is the change in the estimated area, particularly if the designer's goal is to split a large design across two chips. For example, splitting the VT horizontally before x16 results in a two-process design with a small first process, but with a second process using the same number of multiplication and add processors as the single-process design. Since the multiplication processors dominate the area estimate, the second process alone occupies almost as much area the single-process design. If the designer's goal in using Process Creation was to split a large design across two smaller chips, these results are unacceptable. In contrast, splitting the VT horizontally before x26 results in two processes, each of which is approximately half the size of the single-process design, and possibly able to fit onto the chip.

The number of values transferred between the two processes is also of concern, since it relates to the amount of driver circuitry required, and the number of pins on the chip. For the horizontally split designs, this number varies with the location of the process division, although unfortunately the design with the smallest area (split horizontally before x26) also has the largest number of values transferred between the processes. In contrast, the example in the next section divides the two processes vertically, reducing both the per-process area and the inter-process data transfer.

B. An Alternate Two-Process Filter

The previous section described the transformations needed to manually split the filter's VT horizontally into two processes. This section presents results for another two-process design, this time with a vertical division. In Fig. 8, the dark gray oval becomes the second process. The CSTEP control step scheduler and the EMUCS data path allocator synthesis results for this design are shown in the bottom row of Tables I and II.

For the vertical split, Table I shows several effects of Process Creation on the design's total microcode. After the design is split vertically, it still requires 17 cycles to execute, with the first process executing in 17 cycles, and the second process executing concurrently in 10 cycles. However, since the first process alone requires the same number of control steps as the single-process design, the total amount of microcode required for control increases. In contrast, the total number of bits required to control the data path decreases, but not enough to offset the additional amount necessary for control, so the overall effect is an increase in the total amount of microcode.

For the vertical split, Table II shows several effects of Process Creation on the data path. The total number of addition and multiplication processors in the two-process design matches that of the single-process design, meaning that no additional processors were added due to Process Creation. The number of registers and multiplexor input bits increase due to the communication overhead, but only slightly. Since the processors dominate the estimated area, the resulting two-process design requires only slightly more total area than the single-process design. Depending on the layout, this could result in smaller path lengths, a faster clock, and better performance.

C. Comparison of the Results

Comparing the vertically-split two-process design from Section B to the horizontally-split designs of Section A, the vertically-split design requires more total microcode than the single-process design, whereas the horizontally-split designs generally require less microcode. However, the vertically-split design requires only slightly more hardware than the single-process design, and no additional processors, whereas most of the horizontally-split designs require additional hardware.

Another difference between the vertically-split design and the horizontally-split designs lies in the amount of data transferred between the two processes. When the design is split horizontally before x16, 6 values are transferred from the first process to the second, and 3 values are returned, assuming that values used exclusively by one process are not transferred. For the vertically-split two-process design, 3 values are sent from the first process to the second process, and 2 values are returned. If each process was implemented on a separate chip, the vertically-split design would require 44% less inter-chip data transfer, possibly reducing the amount of driver circuitry and external pins.

As stated earlier, these results (reduction in per-process area or microcode) are not guaranteed to occur for all designs, or for all process divisions. However, the idea that the per-process area and the number of values shared between processes can be varied with different process formations follows our intuition. The Workbench, with its behavioral transformations, provides an interactive CAD environment for exploring such alternatives, and a basis for future Algorithmic Level synthesis tools.

VI. Pipestage Creation

The *Pipestage Creation* transformation, acting as a follow-on transformation to Process Creation, converts a set of processes to pipestages. This conversion is done by removing RECV / SEND pairs from the intermediate processes so that the final stage can send its results directly back to the first stage, and by adding the new assumptions that the stages will execute in lockstep with all data transfers between stages occurring at the same time. Examples of Pipestage Creation are presented in the next section.

Pipestage Creation may be applied to any intermediate process in a chain of process vtbodies formed through Vtbody Formation and Pipestage Creation, such that the intermediate process vtbody, V_j, is preceded by vtbody or process vtbody V_i and followed by process vtbody V_k, and SENDs and RECVs the sets of values A, B, C, and D as shown in Fig. 9. V_j receives the sets of values A, B, C, and D from the previous stage V_i, and sends the sets of values C and D to the next stage V_k. V_j receives the set of values C from the next stage V_k, and sends those values along with the set of values A back to the previous stage V_i.

Given an intermediate process of the form specified in the previous paragraph, the following occurs during Pipestage Creation:

1. The final RECV / SEND pair of operators in process V_j are removed.
2. The set of values A (that was sent back to the previous stage V_i, but that was not received from the next stage V_k) is sent to the next stage V_k, and from there to all successive stages until the final stage is reached, where it is sent directly back to the first stage.

   The result of these two steps is shown in Fig. 10.
3. It is assumed that the pipestages will execute in lockstep, and all data transfers between the pipestages will occur at the same time. This assumption may be enforced either through control step scheduling, through added hardware or microcode, or by some other method.

Together with the transformations described earlier, Pipestage Creation allows a designer to interactively

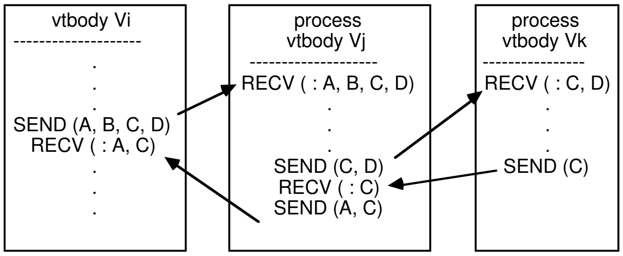

Fig. 9. Before Pipestage Creation

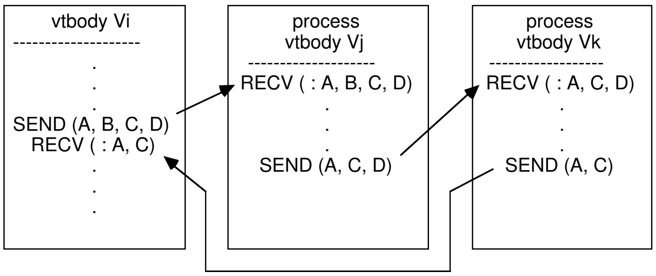

Fig. 10. After Pipestage Creation

pipeline the design's dataflow, choosing the stage boundaries and the operations to be placed in each stage. Thus a wide range of designs can be explored, ranging from designs with functionally decomposed stages (such as most CPUs) to designs with more balanced stages (such as pipelined multipliers or signal processing chips). This differs from the Sehwa system [7], which automatically pipelines the design, but produces only the latter style of pipeline.

Additional work will be necessary before the Workbench can completely synthesize pipelined designs. At the present time, the CSTEP control step scheduler can not schedule pipelined designs, and the EMUCS data path allocator can not allocate the data path for pipelined design. However, research into scheduling pipelined designs, including designs with multiple reservation tables, is currently under way at CMU.

VII. AN EXAMPLE OF PIPESTAGE CREATION – PIPELINING THE 6502

This section presents an example of pipelining, both to demonstrate the utility of the transformations in processing medium-sized designs, and to demonstrate the methods of their application. The design chosen is the MOS Technology MCS6502[4]. The example begins with a "generic" behavioral description that describes the unpipelined behavior of the design. Using the transformations described in the previous sections, this description is transformed to produce two pipelined alternatives: a 4-stage pipeline, and a 2-stage pipeline with a control schedule similar to the commercial design. Other designs, with a different choice of stages, could also be generated from the same starting description.

[4] Throughout this paper, "MCS6502" will be used to refer to the commercial design, and "6502" will be used to refer to the Workbench design.

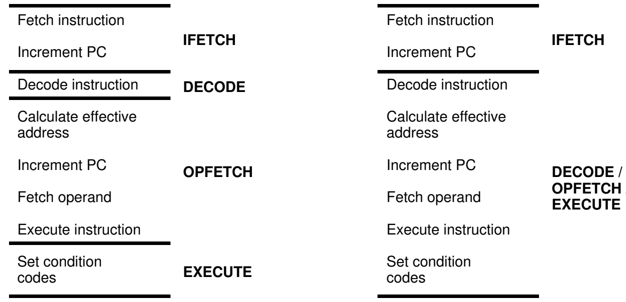

Fig. 11. Alternatives in Pipelining the 6502

At this time, the Workbench has no data path allocator capable of synthesizing pipelined designs, so the results of the transformations can not be evaluated with a completed design. To at least partially evaluate the transformed VT, the CSTEP control step scheduler is used to assign control steps to the design, and individual instructions are "simulated" to determine their execution times. The control steps are also processed slightly by hand to take into account pipelined SELECTs, which are not considered by CSTEP. Since the EMUCS data path allocator does not modify the control steps scheduled by CSTEP, the resulting control step schedule should be a good first-order approximation of the final design's performance.

Two pipelined alternatives for the 6502 are presented, both based on the same 6502 ISPS description. The first alternative, illustrated on the left side of Fig. 11, is the example that was used as the transformations were developed. In this alternative, the design was partitioned into a 4-stage pipeline, with IFETCH, DECODE, OPFETCH, and EXECUTE stages. The second alternative, illustrated on the right side of the figure, is partitioned more like the commercial MCS6502, and divides the design into a 2-stage pipeline, with an IFETCH stage and a DECODE / OPFETCH / EXECUTE stage. Both alternatives are interactively partitioned using the transformations described earlier.

A. A 4-Stage 6502

The initial ISPS description of the 6502 is highly modular, containing many vtbodies and many CALLs, and several levels of instruction decoding — one decoding operation to decode the instruction's "group" (arithmetic, data transfer, etc.), another to decode the opcode, and a third to decode the addressing mode. Although the description is very readable, the implied control structure is inefficient. For example, with a microcoded controller, each of the CALLs will force a microsubroutine jump, as will each of the SELECT operations. If the design was synthesized with this control structure, an estimated 16 cycles would be required to execute an ORA (OR accumulator) immediate instruction.

The Vtbody Inline Expansion transformation can be applied to the design to recursively expand copies of all vtbodies inline, producing one large vtbody. The SELECT

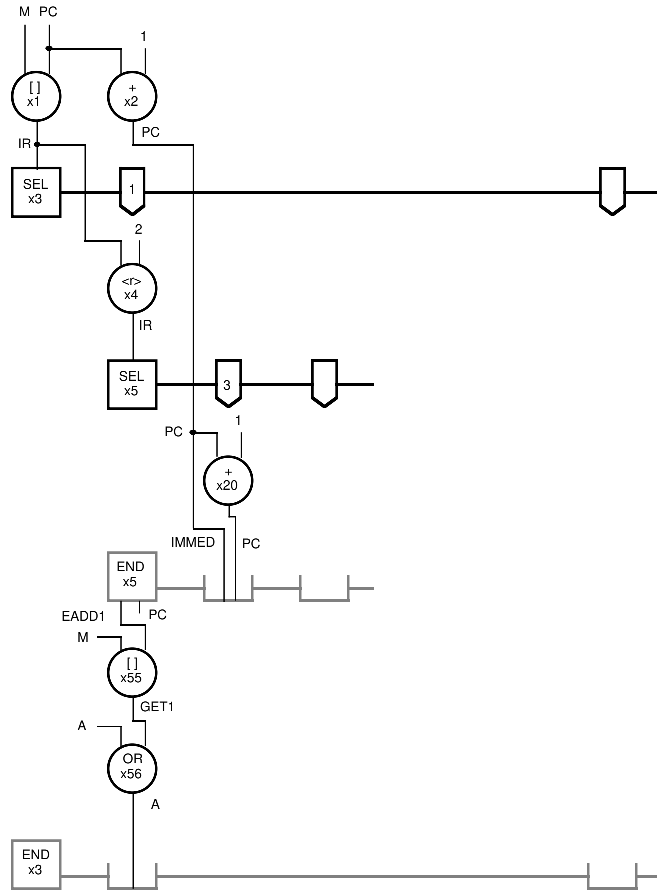

Fig. 12. 6502 Before SELECT Duplication

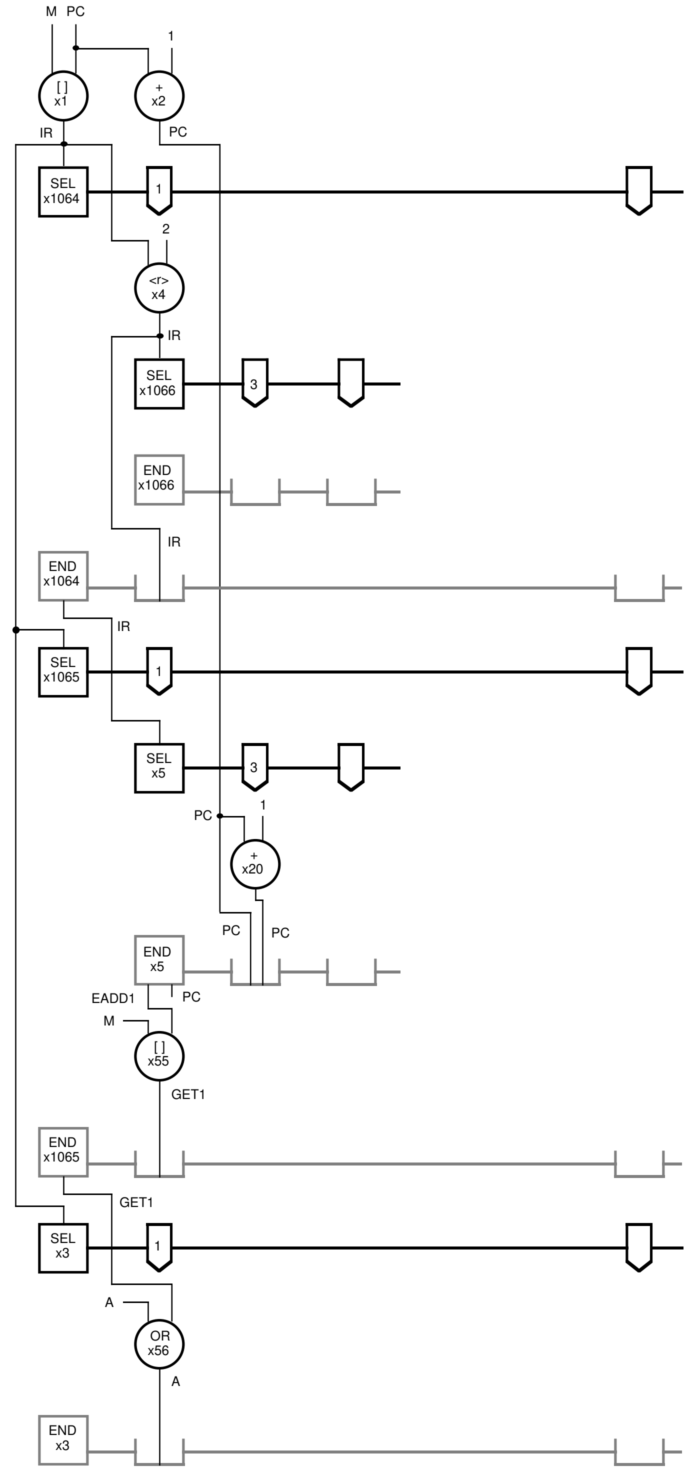

Fig. 13. 6502 After SELECT Duplication and SELECT-To-SELECT Transfer

Motion and SELECT Combination transformations can then be used interactively to reduce the original description's three levels of instruction decoding to two levels — one for the opcode decoding, and one for the addressing mode decoding. After these transformations, the design would require only 6 cycles to execute the ORA immediate instruction.

For this example, the design is partitioned into IFETCH, DECODE, OPFETCH, and EXECUTE stages. First, the SELECT Duplication transformation is used to duplicate the decoding SELECTs to produce copies for each stage other than the IFETCH stage. Then the SELECT-TO-SELECT Transfer transformation is used to move operators between the SELECTs on a branch-by-branch basis. Figs. 12 and 13 show VT diagrams for the ORA immediate instruction before and after these transformations. The other 6502 instructions (not shown) are processed similarly.

Using the Vtbody Formation transformation, each SELECT can then be encapsulated into a new vtbody, forming IFETCH, DECODE, OPFETCH, and EXECUTE vtbodies. Using the Process Creation transformation, the vtbodies are converted into processes by replacing the CALL and LEAVE operators with SEND / RECV message-passing operators, and adding a RESTART operation so that the processes will operate continuously. Finally, using the Pipestage Creation transformation, the processes are converted to pipestages by removing intermediate RECV / SEND pairs; this results in the final stage sending its results directly back to the first stage. The vtbody calling sequences resulting from these three transformations are shown in Fig. 14.

The commercial MCS6502 has a pipelined prefetch, the ORA immediate instruction requires 3 cycles to execute, and a new instruction can be initiated every 2 cycles. In this Workbench design, with 4 pipestages, an estimated 5 cycles are required to execute each ORA immediate instruction, and a new instruction can be initiated every 2 cycles. By partitioning the design into only two stages, and with further SELECT Combination, these times can be reduced to match those of the commercial MCS6502; this transformation process is described in the next section.

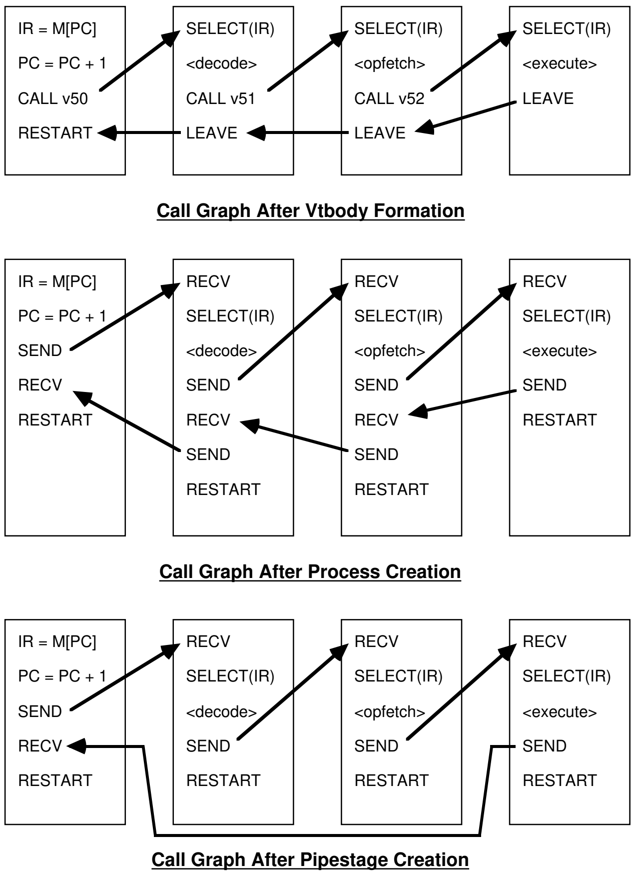

Fig. 14. 6502 Call Graphs

B. A 2-Stage 6502

Beginning with the same unpipelined 6502 ISPS description, a 2-stage 6502 can also be produced, with an IFETCH stage and a DECODE&EXECUTE stage. As before, the Vtbody Inline Expansion transformation can be applied to expand all vtbodies inline to produce a single large vtbody. The SELECT Motion and SELECT Combination transformations can then be applied, this time reducing the original description's three levels of decoding to a single level which decodes both the instruction and the addressing mode. With only these two stages, the SELECT Duplication and SELECT-to-SELECT Transfer transformations are unnecessary, so the Vtbody Formation transformation is used to encapsulate the instruction decoding and execution portion of the VT into a new vtbody, and the Process Creation transformation is used to convert it into a process. Since there are no intermediate stages, the Pipestage Creation transformation is also unnecessary.

The resulting control step schedule for the ORA immediate instruction is shown below. Blank lines separate control steps, and the horizontal line separates the two pipestages. As with the commercial design, the ORA immediate instruction requires 3 cycles to execute, and a new instruction can be initiated every 2 cycles:

```
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
```

Other designs, with a different set of stages, or with the transformations applied differently, can also be created from this same starting description. Thus these transformations allow a designer to quickly explore different pipeline structures, evaluating each in terms of the control schedule.

VIII. Summary

In the past, design automation has focused on the Functional Block, Gate, and Circuit Levels of abstraction. Now that CAD and DA tools have gained acceptance at those levels, it is appropriate to begin developing tools for higher levels of design. At the next higher level, the Algorithmic Level, it is appropriate to explore such alternatives as single-process design versus multiple-process design, and unpipelined versus pipelined design.

This paper has presented a set of transformations that allows a designer to explore these issues in an interactive fashion. Some transformations, in particular the vtbody and SELECT transformations, can be used to transform a design from one based on a readable behavioral description into one with a more efficient control structure. Other transformations, such as the process and pipestage transformations, have a much more extensive effect, dramatically changing the implementation of the design. Transformations similar to those defined in this paper could be defined for other synthesis systems that use dataflow / controlflow graphs, and effects similar to those described here should be observed.

Several examples were presented to demonstrate the use and effects of these transformations. To demonstrate process creation, a description of a digital wave filter was synthesized as a single process, then split into two processes, and synthesized again. Depending on where the design was divided, different effects were observed. For one division, the total microcode decreased by as much as 26%. For another division, the design was split into two data paths that occupied only slightly more area than the single-process design; thus, with a good process division, it is possible that designs that can not fit onto a chip of a given size can be split into two processes, each of which can fit onto a chip of that size.

To demonstrate pipestage creation, an unpipelined 6502 behavioral description was transformed to produce two different designs, one with a 4-stage pipeline, and one with a 2-stage pipeline. For most instructions, the control schedule for the 2-stage design is comparable to that of the commercial MCS6502. Other designs, with a different set of stages, could also be generated from this same starting description.

References

[1] M.R. Barbacci. Instruction Set Processor Specifications (ISPS): The Notation and Its Applications. *IEEE Transactions on Computers* C-30(1):24–40, January, 1981.

[2] G. Borriello and E. Detjens. High Level Synthesis: Current Status and Future Directions. In *Proc. of the 25th DAC,* pp. 477–482. ACM/IEEE, Anaheim, California, June, 1988.

[3] R. Camposano and R. K. Brayton. Partitioning Before Logic Synthesis. In *Proc. of ICCAD-87*, pages 324–326. ACM/IEEE, Santa Clara, California, November, 1987.

[4] P. Dewilde, E. Deprettere, and R. Nouta. Parallel and Pipelined VLSI Implementation of Signal Processing Algorithms. In S.Y. Kung, H.J. Whitehouse, and T. Kailath (editors), *VLSI and Modern Signal Processing*, chapter 15, pages 257–264. Prentice-Hall, 1985.

[5] M.C. McFarland, A.C. Parker, and R. Camposano. Tutorial on High Level Synthesis. In *Proc. of the 25th DAC,* pages 330–336. ACM/IEEE, Anaheim, California, June, 1988.

[6] J.A. Nestor and D.E. Thomas. Behavioral Synthesis with Interfaces. In *Proc. of ICCAD-86*, pages 112–115. IEEE, Santa Clara, California, November, 1986.

[7] N. Park and A.C. Parker. Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications. *IEEE Transactions on CAD* CAD-7(3):356–370, March, 1988.

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

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

[10] D.E. Thomas, C.Y. Hitchcock, III, T.J. Kowalski, J.V. Rajan, and R.A. Walker. Methods of Automatic Data Path Synthesis. *IEEE Computer* 16(12):59–70, December, 1983.

[11] D.E. Thomas, E.M. Dirkes, R.A. Walker, J.V. Rajan, J.A. Nestor, and R.L. Blackburn. The System Architect's Workbench. In *Proc. of the 25th DAC,* pages 337–343. ACM/IEEE, Anaheim, California, June, 1988.

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

[13] R.A. Walker and D.E. Thomas. Behavioral Level Transformation in the CMU-DA System. In *Proc. of the 20th DAC*, pages 788–789. ACM/IEEE, Miami, Florida, June, 1983.

[14] R.A. Walker and D.E. Thomas. A Model of Design Representation and Synthesis. In *Proc. of the 22nd DAC*, pages 453–459. ACM/IEEE, Las Vegas, Nevada, June, 1985.

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

[16] R.A. Walker. *Design Representation and Behavioral Transformation for Algorithmic Level Integrated Circuit Design*. PhD thesis, ECE Dept., Carnegie Mellon University, April, 1988.

Robert A. Walker (S'79–M'88) received the B.S. degree in electrical engineering from Tennessee Technological University, Cookeville, TN, in 1981, and the M.S. and Ph.D. degrees in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, PA, in 1983 and 1988, respectively.

He is currently a PostDoc in the Electrical and Computer Engineering Department at Carnegie Mellon University, where he is researching the use of VHDL as an input language for high-level synthesis. Dr. Walker is a member of the IEEE, ACM, and Sigma Xi.

Donald E. Thomas (S'74–M'77–SM'86–F'89) is a Professor of Electrical and Computer Engineering at Carnegie Mellon University, Pittsburgh, PA. He received his PhD there in 1977, and has worked in the area of automatic design of digital systems since then. From October 1985 through July 1986, he was a Visiting Scientist at IBM T.J. Watson Research Center, Yorktown, NY. Dr. Thomas is Chair of the 1989 Design Automation Conference. He is also a member of the ACM and a fellow of the IEEE.