PSCP: A Scalable Parallel ASIP Architecture for Reactive Systems

Andreas Pyttel, Alexander Sedlmeier, Christian Veith
Siemens AG, Corporate Technology, ZT ME 5
D-81730 Munich, Germany
{Andreas.Pyttel|Alexander.Sedlmeier|Christian.Veith}@mchp.siemens.de

Abstract

We describe a Codesign approach based on a parallel and scalable ASIP architecture, which is suitable for the implementation of reactive systems. The specification language of our approach is extended statecharts. Our ASIP architecture is scalable with respect to the number of processing elements as well as parameters such as bus widths and register file sizes. Instruction sets are generated from a library of components covering a spectrum of space/time trade-off alternatives. Our approach features a heuristic static timing analysis step for statecharts. An industrial example requiring the real-time control of several stepper motors illustrates the benefits of our approach.

1 Introduction

We present a novel approach for the codesign of reactive systems. It employs a scalable ASIP architecture called PSCP (Parallel StateChart Processor), and uses a static timing validation method by exploiting timing constraints of statechart specifications. The PSCP is designed to contain a variable number of process elements. The key to our approach is to fine-tune the architectural parameters and the instruction set generated for a particular application to satisfy all timing constraints.

The essence of reactive systems design has always been to find optimal specification languages, design styles, and analysis methods to guarantee the desired timing properties of an application. In industrial applications, the timing validation methods for reactive Hardware/Software systems are almost always a combination of extensive testing and simulation. Formal approaches to timing and scheduling validation are mostly based on algebraic notations such as Hoare's CSP [17], Milner's CCS [16], or temporal logic. These notations usually require the use of theorem provers. Other approaches include timed or stochastic Petri nets [14], which – like Statecharts – are difficult to analyze, but are less suitable for the specification of complex systems, because of their non-hierarchical nature.

Most hardware/software codesign approaches target the performance optimization of hardware/software systems with additional constraints such as minimization of area or power dissipation [7]. Usually, a specification is partitioned into a set of system components such as processors, code and special purpose hardware [18]. Recently, the ASIP (Application Specific Instruction set Processor) paradigm has opened new ways for the design of reactive systems. ASIPs are microprocessors with customized architectures and instruction sets, which are tailored to the execution of only a certain class of applications. ASIPs usually feature microcoded architectures [8][10], because of their added flexibility over hardwired control. Being useful only for the execution of "a few" programs, it becomes imperative to automate the production of compilers for ASIPs, which are known as retargetable compilers [8][15] (RCs). As most ASIP research efforts are directed towards DSP applications [9][10], RCs use pattern matching techniques or MILP algorithms [6] to generate instruction sets and micro-operations for pipelined data paths aiming at high throughput rates.

Our approach is different in that we aim at the generation of a custom microcontroller optimized for the handling of many simultaneous external events. We give up the limited programmability of existing ASIPs, and optimize hardware and program for one particular reactive application. This eliminates the need for an RC. Our target platform is based on FPGAs [12], which requires special consideration of the limited available hardware resources, and of the attainable system speeds. Our approach detects the critical paths of an application, and applies iterative improvements to the code pieces representing them, thereby altering the hardware architecture. Most optimizations occur on the microinstruction level. The assembler-level instruction set is mostly used to analyze the data-path requirements of an application, and to compute timing estimates. Usually, performance optimizations will result in increased hardware resources, which is compensated by

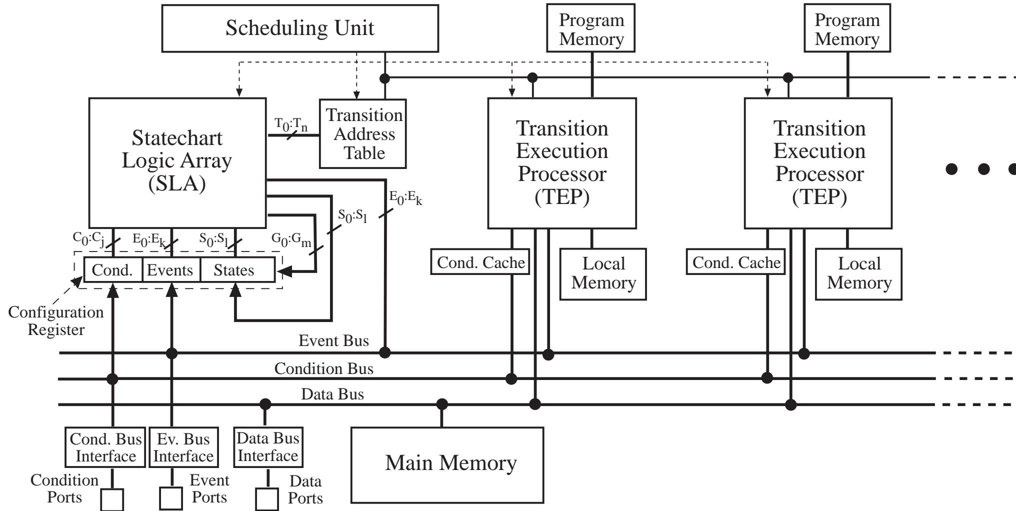

Figure 1: PSCP Architecture Overview

removing unnecessary hardware elements, instructions, and microoperations. In section 2, we describe the notations used in our Codesign system, and briefly review the synthesis of extended statecharts. Section 3 contains a description of our new ASIP architecture. In section 4, we describe our proposed timing validation method, including instruction and architecture selection. In section 5, we present the results we obtained for an industrial application. Section 6 summarizes, and gives an outlook on future research.

2 Extended Statechart Synthesis

Statecharts have been one of the premier specification formalisms for reactive systems for over ten years [2], because of their concise, comprehensive graphical notation. The many features of statecharts include parallel and hierarchical states, and transitions between complex states. However, these unique features also make them difficult to analyze and implement in a hardware/software system. The basic implementation approach, as described in [1], extracts the state and transition information of a chart, and generates a statechart Logic Array (SLA), which implements the semantics of the chart, and acts as a scheduler for the transitions. The SLA concept is illustrated in Fig. 1. The efficient state encoding of a chart involves the generation of exclusivity sets, which was first described in [5]. The state information, together with the encoded events and conditions, forms the configuration register (CR) of the chart (Fig. 1). Its content describes the current state of an application. In [1], the statechart formalism was augmented by external ports for events, conditions, and data. These additions are necessary for hardware/software implementations of statecharts, because a system must be able to react to external events. The external ports (over which external events are delivered) are connected to event and condition buses, which in turn are connected to the CR (Fig. 1). The SLA executes transitions based on the contents of the CR. The SLA generates four sets of outputs: It resets the event parts of the CR (events are only available during a single system cycle), it produces a set of signals for the Transition Address Table, and updates the state part of the CR under the control of the guard signals $G_0..G_m$ (Fig. 1) The guard signals are needed to ensure the correct execution of statecharts [1].

Our new ASIP design system not only uses the graphical language of statecharts, but also introduces additional notations. We define a textual representation for statecharts, which is the starting point of the hardware and software generation process. (Fig. 2a). It is straightforward to generate the textual representation from statechart pictures. We also introduce C as notation for the action parts of transition labels. Thus, function calls are now possible during a transition. Functions can call other functions, but recursion is not permitted. The syntax slightly deviates from C in allowing declarations of the form “int:16” and constants such as “B:001011” to specify bit widths of data elements, events, or ports. Careful range specification helps the ASIP generator to select an optimal architecture. A second aspect of the C notation is its role as an intermediate format between the statechart notation and the assembler-level representation. Fig. 2b shows part of the C code generated for the example presented in section 5. It contains part of a preamble of data types that are always part of the generated C code, plus some port declarations. These code pieces are not actually executed, but

```
basicstate Errstate {
  transition {
    target Idle1;
    label "INIT or ALLRESET/InitializeAll()"
  }
}
andstate Operation {
  contains DataPreparation, ReachPosition;
  transition {
    target Idle1;
    label "INIT or ALLRESET/InitializeAll()";
  }
  transition {
    target ErrState;
    label "ERROR/Stop()";
  }
}
orstate DataPreparation {
  contains OpcodeReady, EmptyBuf, Bounds, NoData;
  default OpcodeReady;
}
```

Figure 2a: Textual statechart format

```
enum ECD {Event, Condition, Data};
enum Encoding {Onehot,Binary};
enum PortDir {Input,Output,Bidirectional};
typedef struct port {
  ECD          Type;
  int:8        Width;
  int:8        Address;
  PortDir      Direction;
} Port;
typedef struct ec {
  ECD           Type;
  int:4         Size;
  int:8         Representation;
  int:4         PositionInPort;
  Port          p;
  int:32        TimeConstraint;
 } EventCondition;

Port PE0={Event,1,0700,Output};
Port CE0={Condition,1,0712,Bidirectional};
Port Buffer = {Data,8,0717,Bidirectional};
EventCondition X_PULSE={Event,1,B1,0,PE0,400};
```

Figure 2b: Intermediate C code

used by the compiler to generate the hardware port architecture, and instruction sequences to access the ports. In the final implementation, a port is represented by an address. However, the C code also contains the action routines written by system designers, which become the executable modules of the final implementation. The C code is generated by a frontend called the Statechart Structural Analyzer, which also generates a BLIF description of the SLA. These two formats are the starting point for the architecture and instruction selection process. The BLIF description is converted to VHDL, and can be immediately synthesized. In total, our system contains two system-level notations (graphical and textual statechart representation), three levels of representation for software (C code, assembler code, and microinstructions), and three formats to represent hardware (PSCP macro blocks, schematics, and VHDL).

3 The PSCP Architecture

3.1 Overview

The main blocks of the PSCP architecture are the SLA and the CR, one or more Transition Execution Processors (TEPs), the Transition Address Table, an overall scheduler, TEP program memory, TEP local memory, main memory, condition caches, and the event/condition bus architecture. An overview of the architecture is given in Fig. 1. The execution of the PSCP is controlled by the scheduler, which enables the SLA at the beginning of a configuration cycle. The SLA generates the addresses of the transitions to be executed according to the statechart description. The scheduler copies the contents of the condition part of the CR into the local condition caches, and assigns the execution of the individual transitions to the available TEPs employing a round-robin protocol. Thus, depending on the number of TEPs, several transitions can be executed in parallel. The TEP receives the trigger signal of the scheduler, picks up a transition address, and executes the corresponding instruction stream. At the end of a transition execution, the scheduler copies the condition cache back to the CR. Transitions are scheduled until the Transition Address Table is empty. The TEPs may generate new events in the CR, and alter the contents of their condition caches, thus generating a new configuration. The scheduler then enables the SLA to begin the next configuration cycle, at which time the new external events are sampled into the CR.

3.2 TEP Architecture

The TEP has a modular and scalable architecture. The basis is a core of elements and a basic instruction set that are necessary for a minimal functional microcontroller. A top-level view of the TEP architecture is shown in Fig. 3. It consists of on-chip RAM, a calculation unit with two registers (an accumulator and a second operand register) and an ALU. In the basic TEP, the databus is 8 bit wide and the instruction format has a width of 16 bit. The instruction set includes load and store instructions, basic arithmetic and logic instructions, shift instructions, jump instructions, and port instructions. Further operations read the transition registers, perform calls to the transition routines, and communicate with the SLA.

To support the execution of statechart models, the TEP has ports for events and conditions, and operations to alter the condition and event registers of the SLA. Further, there are ports for external RAM and for the program memory. To increase flexibility, the TEP has a Harvard architecture [11]. Data-port operations always move a

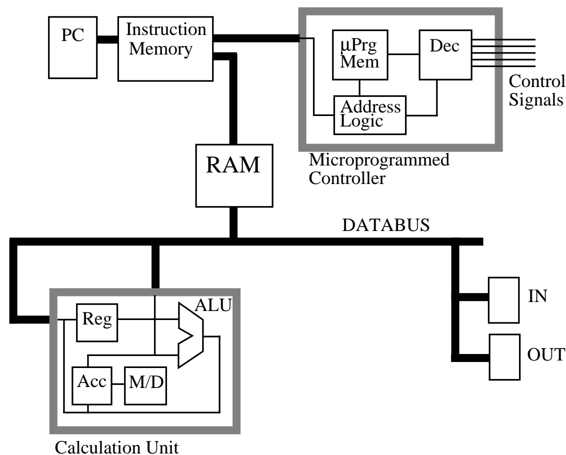

Figure 3: TEP Architecture

complete data word. Conditions and Events vary in size, and need not be uniform for a single application. Every condition or event has a unique address. The implementation of the port architecture, and the port addresses, are generated from the intermediate C description of the application (Fig. 2b).

The control unit of the TEP is implemented in a microprogrammed fashion [11]. Microprogrammed controllers have certain advantages over hardwired controllers: They have a regular structure with medium complexity, and facilitate changes and extensions to the instruction set. On the negative side, a microprogrammed design is slower compared to random logic. In the area of ASIPs, where extensibility is more important than high clock speeds, microprogramming is the technique of choice to design control units [8][9].

Symbolic	Encoding
arithmetic	001 01x00
logical	001 000xx
shift	010 0xxxx
single signals	011 xxxxx
address bus	100 0xxxx
jump, branch	101 0xxxx

Table 1: Microcode format

Each instruction of the TEP is represented by a microprogram containing a sequence of microinstructions. Every microinstruction defines a set of datapath control signals that are asserted in a single state. The required number of control signals determines the bitwidth of the microinstructions. To reduce the bitwidth, signals are encoded, taking advantage of the fact that not all control signals are used by every microinstruction. In the basic TEP, microinstructions are 16 bits wide. The first eight bits represent the control signals, and the other eight bit indicate the address of the next microinstruction. The eight control bits are further divided into 3 bits to denote the group of control signals, and 5 bits to encode the control signals. Currently, there are five groups, which distinguish the different types of microinstructions: ALU instructions, address bus instructions, jump instructions, and a group of instructions that influence exactly one control signal. Table 1 summarizes the format of microinstructions.

3.3 PSCP Modularity

A primary design goal of the PSCP has been easy extensibility and scalability. This holds for the statechart-related aspects as well as for the processor-related aspects of the architecture. The port architecture, the condition/event buses, the SLA, the configuration register and the transition address logic are all variable in size. The generation of these entities does not depend on any special properties of the chart a particular PSCP version was derived from, except for obvious hardware limits such as the number of available pins.

The TEP of an application is derived from a library of elements consisting of hardware building blocks and associated microinstruction sequences. The main library elements are calculation units of varying size and functionality. There are units with or without associated register files, and units with or without shifting capabilities. Several styles of ALUs, which are a subblock of the calculation unit, are available. The library also contains several storage alternatives: Fast, but more expensive registers, moderately fast and moderately expensive internal RAM, and slower, but cheaper external RAM. Simple components such as shifters and registers can be combined to custom operations, which are derived from the assembler code. These instructions execute within one clock cycle. Care must be taken that such instructions do not become the critical paths inside the TEP. This puts a limit on the size of the expressions for which custom instructions may be generated. Additional library elements are available to optimize control structures. Finally, TEPs can be replicated to form PSCP versions with several processing elements in the style of a MIMD machine. The processing elements share the event, condition, and data buses, and the port architecture. Once a particular PSCP version has been fixed, the associated microprogram decoder can be synthesized from the combination of all microinstruction sequences involved.

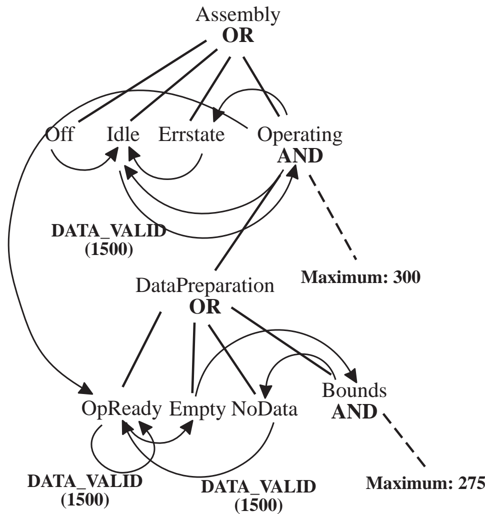

Figure 4: Partial statechart graph

4 Timing validation and instruction selection

Previous researchers have noted that the validation of statecharts, which amounts to reachability analysis, is NP-complete [4], even for basic statecharts. Extended statecharts are at least as difficult to analyze as basic statecharts, because of their additional data dependencies. Although it is conceivable to extend existing state exploration techniques for FSMs to statecharts, we are not aware of any published results. Therefore, we have developed a heuristic algorithm that only partially evaluates chart configurations. Consider Fig. 4, which depicts a part of the statecharts of the example presented in the next section (Fig. 5, Fig. 6). The tree is augmented by the chart´s transitions, resulting in a directed graph. We add timing constraints in the form of arrival periods of external events, in this case “DATA_VALID“, which occurs every 1500 cycles of a reference clock. A perfect algorithm would have to make sure that a DATA_VALID event can be consumed within that time frame from every possible configuration the chart may attain, thus requiring reachability analysis. In practice, however, designers write charts in a modular fashion instead of producing “spaghetti charts“. Therefore, our algorithm localizes the problem by first searching for every state that consumes the desired event in the chart. From there, a depth-first search [13] is started that tries to find event cycles in the graph. An event cycle is a path between two states whose trigger sets both contain the desired event. The result may either be a simple path or a cycle in the graph. The length of an event cycle is defined as the combined length of the transitions in the path. The algorithm must take into account that some transitions that are explored will lead to parallel states. Following every parallel state would lead to combinatorial explosion. Therefore, whenever a parallel substate must be explored, an upper bound is computed for its parallel siblings. In Fig. 4, for every step the algorithm takes in the “DataPreparation“ state, the upper bound of its parallel sibling, in that case 300 cycles, has to be added. The upper bound for a parallel sibling is computed recursively by traversing its associated subtree: At an OR-state, the maximum length transition of this node´s children is computed. At an AND-state, the result is the sum of the length of this node´s children. If possible, the transition lengths are derived from the assembler code of their associated routines, otherwise explicit timing constraints must be specified. The quality of the upper bounds can be improved by careful specification of timing requirements. As a result, the algorithm discovers a list of event cycles, which are compared with the timing requirements of this event.

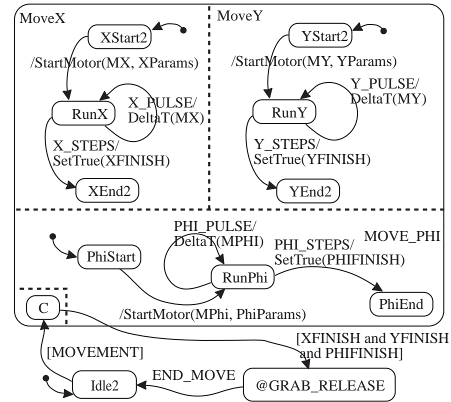

Figure 5: Motor control statechart

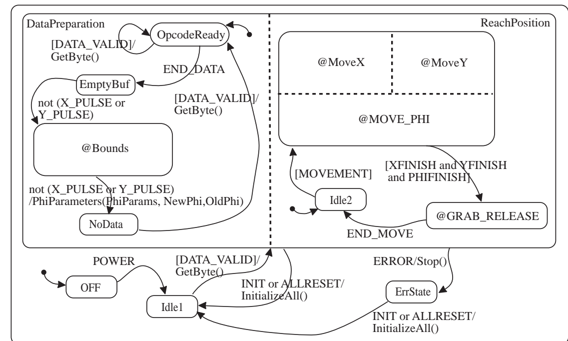

Figure 6: Top-level statechart

If a violation for an event cycle is detected, improvements are applied in increasing order of difficulty to the transitions in question. The optimization steps are performed on a dataflow representation of the microprograms. First, a peephole optimization step removes redundant jumps from the microprogram sequences. Then, the type of storage elements and their associated Load/

Store instructions are changed from external to internal to registers recomputing the timing values for each step. After the simple optimizations, pattern matching is used: If, e.g., a pattern of the form “if (a == b) ... else ...” is detected, a calculation unit with an additional comparator is inserted; if patterns of the form x = -x are detected, an ALU capable of performing two´s complement is inserted. Thus, a number of expressions and control structures can be optimized. The next level are custom instructions for arithmetic expressions found in the transition routines. Complex expressions are broken up into smaller ones not to introduce long critical paths in the design. The last resort is the addition of more TEPs, but this has repercussions on the design of the SLA in the application, because of possible bus contention. Therefore, designers must indicate which transition routines should be mutually exclusive. Then, additional decode logic can be generated so that mutually exclusive routines are not scheduled in parallel. The final set of selected library elements for a PSCP version determines the set of microinstructions needed for the application. The specific microprogram decoder for this application can therefore be easily synthesized.

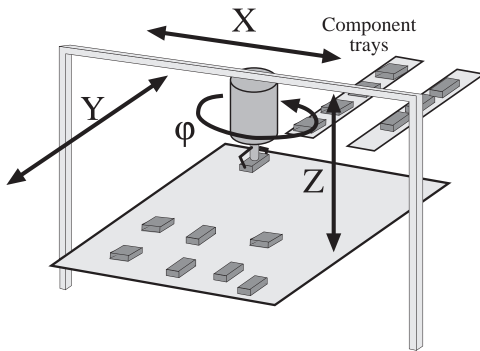

Figure 7: SMD pickup-head

5 Example and results

To demonstrate the benefits of our approach, we modeled the controller of a pickup head for the placement of SMD components on a PCB. The head is part of an automatic SMD assembly systems. An assembly system applies soldering paste to the PCB, places the components, and then heats the board to create the desired electrical connections. A typical assembly machine contains dozens of microcontrollers. Many of them, such as the pickup head controller, drive stepper motors. In our example, four motors have to be controlled that move the head in the x, y, z, and φ coordinates (Fig. 7). The X and Y motors operate with a maximum step frequency of 50kHz, the Z and ϕ motors with 9kHz. One step of the X, Y, and Z motors corresponds to 0.025mm, one step of the ϕ motor leads to an 0.1° rotation. The maximum velocity of the X and Y motors are 1.25m/sec, their maximum acceleration is 10m/sec^2. The maximum x and y distance of an assembly head movement is 1m in each direction. The statechart for the head-positioning part of the application is shown in Fig. 5. The motors are set in motion by counters that issue a pulse on zero. The Z and ϕ motors move uniformly, while the X and Y motors have to be accelerated and decelerated in a precise way, because of inertia. For a 15MHz reference clock, this leads to timing requirements of 300 cycles to update the X and Y counters. Further, the controller can receive commands from a central controller every 1500 cycles (Table 2). The top-level chart of the complete application is shown in Fig. 6.

Event	Cycles
DATA_VALID	1500
X_PULSE	300
Y_PULSE	300
PHI_PULSE	1600

Table 2: Timing Constraints

Cycle	Length
{Idle1, ReachPosition, Idle1}	235
{OpReady, OpReady}	747
{Idle1, OpReady}	105
{OpReady, EmptyBuf, Idle1}	772
{OpReady, EmptyBuf, Bounds, Idle1}	1414
{OpReady, EmptyBuf, Bounds, NoData}	**2041**
{NoData, OpReady}	747
{NoData, Idle1}	130
{NoData, ErrState, Idle1}	180
{RunX, RunX}	**878**
{RunY, RunY}	**878**
{RunPhi, RunPhi}	878

Table 3: Event Cycles

The event cycles detected by the timing validation algorithm are depicted in Table 3. They indicate a possible timing violation for the first three timing constraints of Table 2. Iterative improvement of this example led to an architecture with two TEPs, calculation units with extra multiply/division capability, a 16 bit wide data bus, and additional registers. The solution fulfils all timing requirements. Timing and area results are summarized in Table 4. The result fits on a single Xilinx® XC4025 FPGA, which contains 1024 CLBs [12]. The floorplan of the result is shown in Fig. 8.

6 Conclusion and future work

In this paper, a flexible ASIP architecture was presented, which is suitable for the implementation of reactive systems. The architecture contains special elements for the efficient execution of statechart models. It is scalable with respect to the number of processing elements, and allows the generation of MIMD style machines. A heuristic algorithm for the static analysis of extended statecharts was presented. The benefits of the timing analysis and instruction selection methods were demonstrated with an industrial example.

Architecture	Area	Crit. Path X, Y	Crit. Path DATA_VALID
1 minimal TEP	224	> 1000	> 3000
16bit M/D TEP, unoptimized code	421	878	2041
16bit M/D TEP, optimized code	421	524	1317
2 16bit M/D TEP, unoptimized code	773	469	1081
2 16bit M/D TEP, optimized code	773	**282**	**699**

Table 4: Area and Timing Results

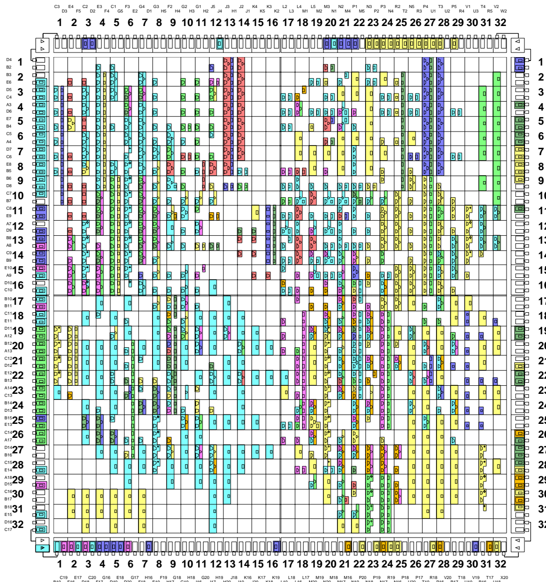

Figure 8: PSCP floorplan

Future work will include pipelined versions of the PSCP architecture, as well as the addition of timers and interrupt capabilities. Further, we will improve the code generation and instruction set selection process by refining and extending the patterns used by the code generator.

7 References

[1] K. Buchenrieder, A. Pyttel, Ch. Veith: Mapping Statechart Models onto an FPGA-Based ASIP Architecture, in proceedings of EURO-DAC'96, 1996.

[2] D. Harel: Statecharts: A Visual Formalism for Complex Systems. Sci. Comp. Prog., vol. 8, pp 231-274, 1987.

[3] N. Binh, M. Imai, A. Shiomi, A New HW/SW Partitioning Algorithm for Synthesizing the Highest Performance Pipelined ASIPs with Multiple Identical FUs, in proceedings of EURO-DAC'96, 1996.

[4] D. Drusinsky: On Synchronized Statecharts. PhD Thesis, Dept. of Comp. Sci., The Weizmann Inst. of Sci., 1988.

[5] D. Drusinsky-Yoresh: A State Assignment Procedure for Single-Block Implementation of State-charts. IEEE Trans. on CAD, vol. 10, No 12, December 1991.

[6] A.Y. Alomari, M. Imai, J. Sato, N. Hikichi: An integer programming approach to the instruction set selection problem, IEICE Trans. Fundam. Electr. Commun. Comput. Sci. (Japan), vol. E76-A, no. 10, p. 1849-57, Oct. 1993.

[7] D. Gajski, F. Vahid, J. Gong: A Binary-Constraint Search Algorithm for Minimizing Hardware During Hardware-Software partitioning, Proc. Euro-DAC, 1994.

[8] R. Leupers, P. Marwedel: Instruction set extraction from programmable Structures, in proceedings of EURO-DAC '94 with EURO-VHDL '94, p. 156-6.

[9] C. Liem, T. May, P. Paulin: Instruction set matching and selection for DSP and ASIP code generation, Proc. of the EDAC-ETC-EUROASIC, Paris, France, 1994, p. 31-37.

[10] J. Van Praet, G. Goossens, D. Lanneer, H. De Man: Instruction set definition and instruction selection for ASIPs, Proc. of the Seventh International Symposium on High-Level Synthesis, p. 11-16, 1994.

[11] D. A. Patterson, J. L. Hennessy: Computer Organization and Design, The Hardware / Software Interface, Morgan Kaufmann Publishers, Inc., San Francisco, CA.

[12] Xilinx, Inc.: The Programmable Logic Data Book, San Jose, CA, 1994

[13] R. Sedgewick: Algorithms in C++, Addison-Wesley, 1992.

[14] J. Tsai, S. Yang, Y.-H. Chang: Timing Constraint Petri Nets and Their Application to Schedulability Analysis of Real-Time System Specifications, IEEE Trans. Softw. Eng., vol. 21, no 1, pp. 32-49.

[15] R. Leupers, P. Marwedel: Instruction set modeling for ASIP code generation, Proc. Ninth Int. Conf. on VLSI Design, IEEE Comp. Soc. Press, pp. 77-80, 1995.

[16] R. Milner: A Calculus of Communicating Systems, Lecture Notes in Comp. Sci. 92, Springer, 1980.

[17] C. A. R. Hoare: Communicating Sequential Processes, Prentice Hall, 1985.

[18] R. Ernst, J. Henkel, T. Benner: Hardware/Software Cosynthesis for Microcontrollers, IEEE Des. & Test of Computers, Dec. 1993, pp 64-75.