# Synthesis and Analysis of an Industrial Embedded Microcontroller

Ing-Jer Huang, Li-Rong Wang, Yu-Min Wang
Institute of Computer and Information Engineering
National Sun Yat-Sen University
Kaohsiung, TAIWAN 804
R. O. C.
EMAIL: {ijhuang,lrwang,ymwang}@cie.nsysu.edu.tw

## Abstract

*This paper presents a case study of synthesis and analysis of the industrial embedded microcontroller HT48100, using the hardware/software co-synthesis tool (PIPER-II) for microcontrollers/microprocessors. The synthesis tool accepts as input the instruction set architecture (behavioral) specification, and produces as outputs the pipelined RTL designs with their simulators, and the reordering constraints which guide the assembler how to generate code for the synthesized designs. The study shows that the synthesis approach was able to help the original design team to evaluate their design quality, analyze the architectural properties and explore possible architectural improvements and their impacts in both hardware and software. Feasible future upgrade for the microcontroller family is identified by the study. Further cooperation with the design team has been undertaken to integrate the synthesis methodology into their design flow.*

## 1. Introduction

Due to the time-to-market pressure and the lack of appropriate CAD tools and design methodology, designers of industrial embedded controllers often miss the opportunity to systematically analyze the architectural properties of their designs and explore hardware and software alternatives for future upgrades. Therefore, cooperation with the design team of the HT48100 embedded microcontroller [1] has been established to investigate the adoption of our system/high level synthesis tools and design methodology into their design flow. In this paper, we will present the preliminary results of the case study of applying our synthesis system PIPER-II to solve the following architectural issues related to the HT48100 microcontroller:

- *How good is the current implementation?* The answer will help the designers and manager assess their design skills and evaluate their product quality.
- *Is there any room for further improvement?* How about different pipeline organizations? How about using various data path components?
- *Does the hardware change impact assembly code (application software)?* Should existing assembly code be repaired/rewritten for hardware upgrades? If yes, how?

The rest of the paper is organized as follows. Section 2 introduces the PIPER-II synthesis system. Section 3 presents the synthesis process of the microcontroller and the results. Section 4 provides the simulation results. Section 5 discusses the architecture exploration. Section 6 provides some concluding remarks.

## 2. Overview of the PIPER-II synthesis system

PIPER-II is the behavioral domain synthesis tool of ADAS, a full-range design automation system for microprocessors [9]. PIPER-II accepts the abstract specification of an instruction set architecture (ISA), and produces pipelined register-transfer level (RTL) designs, consisting of both data and control paths. The tool also generates a reordering table which guides the compiler back-end (*reorderer*), provided by the designers, to properly reorder the instruction stream sequence in order to avoid possible pipeline hazards and improve the pipeline throughput. In addition, PIPER-II provides simulators for the synthesized designs at the ISA, abstract finite state machine, and pipelined RTL levels, respectively. Application benchmarks are provided by the designer to verify and evaluate the synthesized designs.

Figure 1 illustrates the conceptual structure of the PIPER-II system. There are two major design flows in PIPER-II. At the left side is the synthesis flow, and at the right side is the simulation flow.

† The research is partially supported by Holtek Microelectronic Inc., a semiconductor company in Taiwan specialized in design and manufacturing of ASICs and microcontrollers.

ASP-DAC '97
0-89791-851-7$5.00 

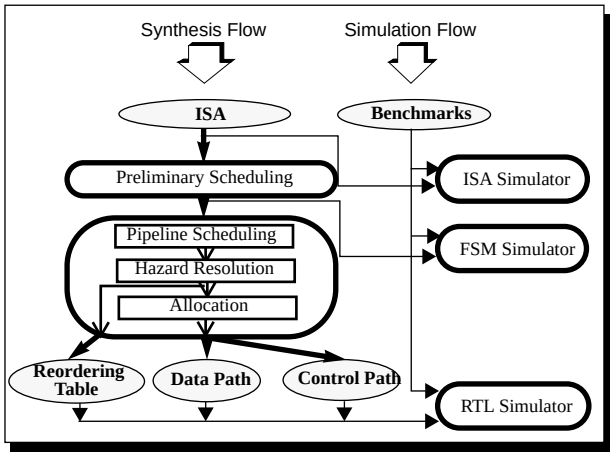


Figure 1. The structure of PIPER-II system

In the synthesis flow, the preliminary scheduling phase translates the ISA specification in Prolog into an abstract finite state machine. The abstract finite state machine is a sequential implementation of the ISA specification, with each state taking one clock cycle for execution. The hardware resource hasn't been allocated for the abstract finite state machine yet. The abstract finite state machine is then turned into a pipelined RTL design by going through the following tasks: (1) pipeline scheduling, (2) pipeline hazard resolution, and (3) resource allocation. The pipeline scheduling phase assigns micro-operations into pipeline stages. Pipeline hazards may be introduced by the pipeline scheduler in highly pipelined cases. These hazards are resolved in the hazard resolution phase by a combination of hardware and software resolution strategies. In this phase PIPER-II generates a reordering table consisting of reordering constraints which instruct the reorderer to properly organize the assembly code for the synthesized pipelines. At the last phase, the hardware resources are allocated, producing a pipelined RTL level design.

Parallel with the synthesis flow is the simulation flow. Designs at various abstraction levels are simulated, using the application benchmarks provided by the designer, to verify the synthesis results and assess the performance/cost tradeoff. In our current implementation, designs at the architecture, abstract finite state machine (scheduled CDFGs with sequential semantics), and pipelined RTL levels are simulated.

PIPER-II is the extended version of our previous synthesis system PIPER [5] [6]. The extensions include the simulation flow and capability of synthesis for parallel processes.

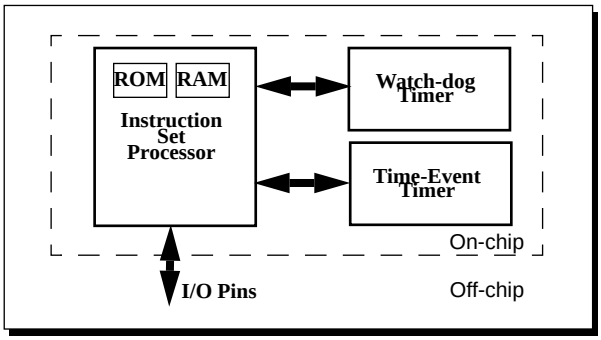


Figure 2. HT48100 system architecture

## 3. Synthesis of the HT48100 embedded microcontroller

### 3.1. The HT48100 system architecture

The HT48100 embedded microcontroller is an high performance RISC-like microcontroler specifically designed for I/O control applications, such as remote controller, fan/light/washing machine controllers, *etc.* [1] Figure 2 depicts the system architecture. The microcontroller consists of an instruction set processor, a time/event timer, and a watch-dog timer. The instruction set processor has 64 instructions, with the instruction word width of 14 bits. The instruction fetch and execution phases both take four clock cycles. In the current implementation, there are two pipeline stages, corresponding to the fetch and execution phases, respectively. The pipeline firing latency is four clock cycles.

The processor has embedded memories: one program ROM and one data RAM. The processor communicates with off-chip world with 18 I/O ports. The I/O ports are mapped to specific data memory (RAM) locations. The processors also communicates with the on-chip time/event timer and watch-dog timer by reading/writing to specific data memory locations as well. The two timers operate autonomously and are responsible to monitoring the system status. They may change the control flow of the instruction set processor by generating interrupts to the processor.

### 3.2. Specification and synthesis of the instruction set processor

As depicted in Figure 2, there are three loosely coupled concurrent components in the microcontroller, with the instruction set processor as the major component that determines most of the performance/cost tradeoffs of both hardware and software. We began the investigation by first synthesizing the instruction set processor, which are presented in the paper. Synthesis of the entire system, including the two timers and their interfaces, will be reported in our future publication.

Shown in Figure 4 is the instruction set architecture specification of the microcontroller, written in Prolog. The specification is purely behavioral, no timing information. The clauses ht, fetch and execute are defined by the designer. The clause ht specifies the major loop body for instruction fetch-and-execution. The clauses fetch and execute specify the behavior of the instruction fetch phase and the instruction execution phase, respectively. Note that the '_sim' clause is a simulation directive and is not part of the behavior specification. This directive instructs the simulator to print out the required information. The built-in class and attributed clauses are used by the designer to declare the (library) types of registers and memory, and the type assignment of the architected registers and memory, respectively. The complete specification consists of about 400 lines of Prolog code.

The original HT48100 microcontroller has two pipeline stages, with the pipeline firing latency of four. So we first synthesized a pipelined RTL design with the same pipeline firing latency and data path library, in order to compare it with the original design. The results and comparisons are highlighted as follows. Synthesized designs with other pipeline firing latencies are reported in Section 5.

- *Data path*: Shown in Figure 3 is the block diagram of the synthesized data path. There is one ROM (two-cycled access), one RAM (single-cycled access), one stack, and one ALU. It uses the same number of data path modules as the original design.
- *Control path*: The control path has 135 states and 60 Boolean functions (controlling 60 signals). The pipeline behavior and instruction timing are the same as the original design.
- *Software*: For the synthesized design, PIPER-II automatically generated the requirement that the branch-related instruction `sz` (and another 12 branch-related instructions) be separated from its (logically) next instruction by one instruction cycle (one pipeline stage), due to potential pipeline hazards. The requirement conforms to the original design. In the original design, the requirement is satisfied by stalling the pipeline once the aforementioned instruction is decoded. The stalled cycles result in performance degradation. On the other hand, PIPER-II is able to generate two

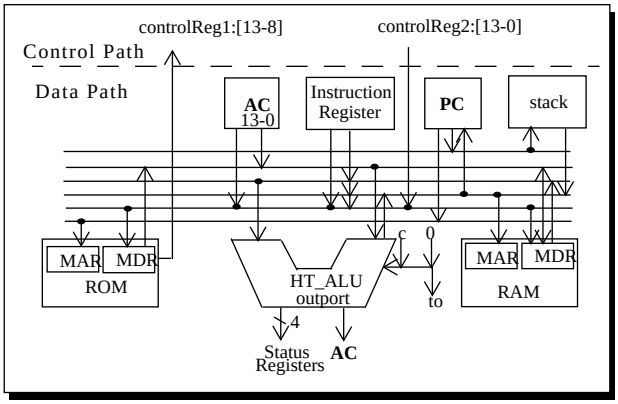


Figure 3. Block diagram of the synthesized design fo HT48100

```
ht( PC, _, AC, TO, PD, OV, Z, ACF, C,STACK, Memory ):-
      fetch( PC, IR, Memory, Op ),                    % Main loop body
      P1 is PC + 1,
      '_sim'(print, [PC, IR ,AC, TO, PD, OV, Z, ACF, C,STACK, Memory]),
                                                      % simulation directive:
                                                      % print out the system state
      execute( [Op], IR, AC, TO, PD, OV, Z, ACF, C, P1, STACK, Memory, IR_next,
        AC_next, TO_next, PD_next, OV_next, Z_next, ACF_next, C_next, PC_next,
        STACK_next, Memory_next ),
      ht( PC_next, IR_next, AC_next, TO_next, PD_next, OV_next, Z_next, ACF_next,
        C_next, STACK_next, Memory_next ).
      % Loop implemented as a tail recursion

fetch( PC, [Op,Adr], Memory, Op ):-
      memory_read( PC, Memory, [Op,Adr] ).

execute( [adc], IR, AC, TO, PD, _, _, _, C, PC, STACK, Memory, IR, AC_next, TO, PD,
   OV_next, Z_next, ACF_next, C_next, PC, STACK, Memory ) :-%Actions of inst.
      MemAR <- [constant(0)/10-7,IR/6-0],ADC
      memory_read( MemAR, Memory, MemDR ),
      [AC_next,TO_next,PD_next,OV_next,Z_next,ACF_next,C_next] <~ AC +
        MemDR + C.

execute( [adcm], IR, AC, TO, PD, _, _, _, C, PC, STACK, Memory, IR, AC, TO, PD,
   OV_next, Z_next, ACF_next, C_next, PC, STACK, Memory_next ) :-%Actions of
   inst.
      MemAR <- [constant(0)/10-7,IR/6-0],ADCM
      memory_read( MemAR, Memory, MemDR ),
      [MemDR_next,TO_next,PD_next,OV_next,Z_next,ACF_next,C_next] <~ AC +
        MemDR + C,
      memory_write( MemAR, MemDR_next, Memory, Memory_next).
      :
      :
      :
      :
      :

% Define properties of module types.

class([shftreg, master_slave,r1,14]).
class([rsincreg,master_slave,r2,11]).
class([reg,     master_slave,r3, 1]).
class([memory,  ram,         m1,[14,2048]]). % 14bit x 2K

% Assign the component types to architected registers and memory
attributed([r2,pc, r,ht,1]).
attributed([r1,ir, r,ht,2]).
attributed([r1,ac, r,ht,3]).
attributed([r3,to, r,ht,4]).
attributed([r3,pd, r,ht,5]).
attributed([r3,ov, r,ht,6]).
attributed([r3,z,  r,ht,7]).
attributed([r3,acf,r,ht,8]).
attributed([r3,c,  r,ht,9]).
attributed([r2,stack,  r,ht,10]).
attributed([m1,m,  m,ht,11]).
```

Figure 4. Instruction set architecture specification for HT48100 embedded controller

designs with different hardware/software tradeoffs: one uses the same stalling approach, and the other uses the *reordering* approach, in which the pipeline is not stalled, but the compiler backend is responsible to reorder other independent instructions or `NOP` instructions into the slots between the aforementioned instructions and their (logically) next instructions. The reordering approach yields better performance, but may require the recompilation of existing code and expand the code size slightly.

In summary, for the given pipeline firing latency and data path library, the original design can be considered as being equivalent to the synthesized design in terms of data path size, pipeline behavior and software compatibility. Based on the observation that the synthesized design is the best solution after numerous design points have been explored by PIPER-II, we concluded that the design team has done a fairly good job in implementing the HT48100 microcontroller. However, PIPER-II shows that performance can be further improved if the reordering approach is adopted.

## 4. Automatic simulation

The instruction set architecture specification in Figure 4 is an executable specification, which can serve as the instruction level simulator. The instruction level simulation observes the system status, including architected registers and memory image, change after the execution of each instruction in the application benchmark. Figure 5 (a) is an "insertion sort" benchmark in HT48100's assembly code. The simulation is invoked by instantiating the initial values of architected registers and memory image in the `ht` clause, as in Figure 5 (a). Figure 5 (b) lists the simulation trace of the first few executed instruction of the benchmark.

The next level of simulation is for the abstract finite state machine, a sequential (microoperation-level) implementation of the given instruction set architecture. The simulator shows the cycle-by-cycle execution of the microoperations. For HT48100, the instruction fetch phase takes four clock cycles, and the instruction execution phase takes from one to four cycles.

The last level of simulation is for the synthesized pipelined data path and control path. In simulation, designers can observe state activities in pipeline stages, registers and memory values and status of functional units' input/output ports, all at the cycle-by-cycle level. Figure 6 shows a segment of simulation trace for the synthesized design. Here we show the first eight clock cycles of the "insertion sort" benchmark. Since each pipeline stage takes four clock cycles, there is a sequence counter (counting from zero to three) in the control path in order to sequence through the four cycles. In the first four cycles (clock = 0 ~ 3), the processor is fetching its first instruction `move`. There is only one instruction in the pipeline. So there is only one active state during each clock; e.g., the state `bc(4)` is active in `stage(1)` at clock = 3. In the next four cycles (clock = 4~ 7), the processor has filled the pipeline, so there may be two active states during each clock (one active state for each pipeline stage). For example, during clock = 5, the states `bc(2)` and `bc(96)` are active in `stage(1)` and `stage(2)`, respectively.

In summary, the simulators have been found very helpful in debugging and verifying the behavior and timing of the synthesized design. In addition, the simulators can be delivered to the potential customers of the processor to evaluate the processor or integrate the simulators into their system simulation for the system they are going to build around the processor.

## 5. Architecture exploration

In Section 3 we showed that performance can be improved if the reordering approach is adopted into the original design. In this section we examine more interesting architecture alternatives by synthesizing the processor with different pipeline firing latencies and library components.

Figure 7 lists the pipeline performance/cost and instruction pair reordering constraints of the these architecture alternatives. HT_8 is the non-pipelined implementation of the processor and can be regarded as a lowest cost design. HT_4 is the synthesized design reported in Section 3, which is equivalent to the original design in terms of hardware size and software compatibility. HT_3 and HT_2 are synthesized with the pipeline firing latencies of three and two (using the same library as the original design), respectively. HT_RD_3 is synthesized with the same latency as HT_3's, but with a faster ROM with one-cycle assess time (the original design used a two-cycled ROM), and a manually performed optimization technique which sets the data memory address register during the decode stage, regardless of the result of the decoding. Comparisons between these designs are summarized as the following.

- Compared with HT_8, HT_4 uses the same number of resources of major data path components and has a slightly larger control path (with five extra boolean functions). With the same data path size and a few extra control signals in the control path, HT_4 provides almost two times faster of pipeline throughput. The two-fold performance improvement come at the cost of software: thirteen dependent instructions need stalling (in which case, performance will be slightly degraded) or reordering.

- HT_3 operates 33% faster than HT_4, with the same data path resource numbers and 38% larger in the control path. The performance improvement also comes at the cost of software: two more instruction pairs need hardware stalling or software reordering, and the reordering distances grow from one to two for eight of the original instruction pairs. If the reordering approach is adopted, then some moderate amount of recompilation of the existing code is necessary. If the stalling approach is adopted, then recompilation is unnecessary.
- HT_2 operates 100% faster than HT_4, at the cost of much larger data path and control path. Furthermore, the number of instruction pairs that need stalling or reordering grows to 315, and the reordering distances also increase significantly. Tremendous amount of recompilation is necessary to retarget the original code to HT_2.
- HT_RD_3 offers the same performance improvement as HT_3 (33% faster than HT_4), but with a 22% smaller control path. The best advantage offers by HT_RD_3 is the full software compatibility with the original design.

In summary, the exploration suggests that the best upgrade path of the processor for the company is to adopt the HT_RD_3 design (faster ROM + earlier MAR setting + pipeline firing latency of three) which offers 33% faster pipeline throughput and full software compatibility. If the faster ROM can not be used, then HT_3 with hardware stalling can be considered as the performance upgrade candidate. However, HT_2 is by no means an ideal upgrade candidate since too many dependent instruction pairs need attention.

## 6. Conclusions

We have presented the industry's need for an appropriate design tools and environment for embedded controller design, which consists of hardware and software components. A pipeline synthesis system PIPER-II has been outlined and its application in the synthesis and exploration for the industrial embedded microcontroller HT48100 has been demonstrated. The study shows that the synthesis approach was able to help the original design team to evaluate the design quality, analyze the architectural properties and explore possible architectural improvements and their impacts in both hardware and software. Feasible upgrade path for the microcontroller family has been identified in the experiment. In addition, simulators are automatically generated for the synthesized designs, which are very helpful for debugging (for the designer) and system development (for the system developers who build their systems with the microcontrollers). Further cooperation with the design team has been undertaken to integrate the synthesis methodology into their design flow.

## Reference


[1] HT48100 Development Data Book, Holtek Microelectronics Inc., Dec. 1994

[2] Mauricio Breternitz Jr. and John Paul Shen, "Architecture Synthesis of High-Performance Application-Specific Processors", *Proc. 27th DAC*, 1990

[3] Richard Cloutier, *Synthesis of Pipelined Instruction Set Processors*, Ph.D. dissertation, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, 1993. Also available as a Research Report No. CMUCAD-93-03.

[4] John Hennessy and Thomas Gross, "Postpass Code Optimization of Pipeline Constraints," *ACM Tran. on Programming Languages and Systems*, July 1983, pp. 422-448

[5] Ing-Jer Huang and Alvin Despain, "High Level Synthesis of Pipelined Instruction Set Processors and Back-End Compilers," *Proc. of 29th DAC*, June, 1992

[6] Ing-Jer Huang, *Co-Synthesis of Instruction Sets and Microarchitectures*, Ph.D. dissertation, Dept. of Electrical Engineering - System Division, University of Southern California, 1994.

[7] Peter M. Kogge, *The Architecture of Pipelined Computers*, MacGraw-Hill, 1981

[8] Nohbyung Park and Alice C. Parker, "Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications," *Trans. on CAD, Vol 7. No. 3*, March 1988

[9] Iksoo Pyo, et al., "Application-Driven Design Automation for Microprocessor Design," *Proc. 29th DAC*, June, 1992

[10] R. G. G. Cattell, "Automatic Derivation of Code Generators from Machine Description," *ACM Trans. on Programming Languages and Systems*, Vol. 2, No. 2, April 1980

[11] S. L. Graham, "Table-Driven Code Generation," *IEEE Computer*, August 1980

[12] Robert Giegerich, "A Formal Framework for the Derivation of Machine-Specific Optimizers," *ACM Trans. on Programming Languages and Systems*, Vol. 25 No. 3, July 1983

```
run:-
ht(100,[],0,0,0,0,0,0,0,0,
% Initial values for architected registers

[(iar,0),          % Initial memory image
 (mp,0),
 (start,1),
 (count,11),
 (idx1,1),
 (idx2,0),
 (temp,0),
 (zero,0),
 (99,0),
 (0,10),
 (1,9),
 (2,8),
 (3,7),
 (4,6),
 (5,5),
 (6,4),
 (7,3),
 (8,2),
 (9,1),
 (10,0),
 (100,[mov,idx1]),
 (101,[sub,count]),
 (102,[snzi,99]),
 (103,[jmp,129]),
 (104,[deca,idx1]),
 (105,[movm,idx2]),
 (106,[sub,zero]),
 (107,[szi,99]),
 (108,[jmp,127]),
 (109,[mov,idx2]),
 (110,[movm,mp]),
 (111,[mov,iar]),
 (112,[movm,temp]),
 (113,[inc,mp]),
 (114,[mov,iar]),
 (115,[sub,temp]),
 (116,[snzi,99]),
 (117,[jmp,127]),
 (118,[mov,iar]),
 (119,[dec,mp]),
 (120,[movm,iar]),
 (121,[mov,temp]),
 (122,[inc,mp]),
 (123,[movm,iar]),
 (124,[dec,idx2]),
 (125,[mov,idx2]),
 (126,[jmp,106]),
 (127,[inc,idx1]),
 (128,[jmp,100]),
 (129,[halt,0])]).
```

(a)

```
Current State :
      PC(100), IR([mov,idx1]), AC(0), TO(0), PD(0), OV(0),
      Z(0), ACF(0), C(0),                /* AC <- idx1 = 1    */
      STACK(0), Memory(
      [(iar,0),(mp,0),(count,11),(idx1,1),(idx2,0),(temp,0),(zero,0
      ),(99,0),
    (0,10),(1,9),(2,8),(3,7),(4,6),(5,5),(6,4),(7,3),(8,2),(9,1),(10,0
      ),...])

Current State :
      PC(101), IR([sub,count]), AC(1), TO(0), PD(0), OV(0),
      Z(0), ACF(0), C(0),                /* AC <- AC - count = 11
      */
      STACK(0), Memory([...])

Current State :
      PC(102), IR([snzi,99]), AC(-10), TO(0), PD(0), OV(0),
      Z(0), ACF(1), C(1),                /* if C != 0 then pc++ */
      STACK(0), Memory([...])
```

(b)

Figure 5. (a) An “insertion sort” benchmark for the instruciton level simulator; (b) Instruction level simulation trace

| Clock | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Sequence Counter | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 |
| Active State | bc(1),stage(1) | bc(2),stage(1) | bc(3),stage(1) | bc(4),stage(1) | bc(1),stage(1)<br>bc(109),stage(2) | bc(2),stage(1)<br>bc(110),stage(2) | bc(3),stage(1)<br>bc(111),stage(2) | bc(4),stage(1) |
| pc | 101 | 101 | 101 | 101 | **102** | 102 | 102 | 102 |
| ir | X | X | X | [mov,idx1] | [mov,idx1] | [mov,idx1] | **[sub,count]** | [sub,count] |
| memAR(1) | 100 | 100 | 100 | 100 | **101** | 101 | 101 | 101 |
| memDR(1) | X | X | [mov,idx1] | [mov,idx1] | [mov,idx1] | [mov,idx1] | [mov,idx1] | [mov,idx1] |
| memAR(2) | X | X | X | X | **idx1** | idx1 | idx1 | idx1 |
| memDR(2) | X | X | X | X | X | **1** | 1 | 1 |
| ac | X | X | X | X | X | X | **1** | 1 |
| : | : | : | : | : | : | : | : | : |

Figure 6. Cycle-by-cycle simulation of the synthesized pipelined RTL design

| Design | HT_8 | HT_4 | HT_3 | HT_2 | HT_RD_3 (1 cycle ROM, earlier MAR setting) |
|---|---|---|---|---|---|
| Instruction Firing latency | 8 | 4 | 3 | 2 | 3 |
| Pipeline stages | 1 | 2 | 3 | 4 | 2 |
| ALU # | 1 | 1 | 1 | 2 | 1 |
| Memory Port # | 2 | 2 | 2 | 3 | 2 |
| Boolean function # | 63 | 68 | 94 | 100 | 73 |
| state # | 170 | 170 | 170 | 171 | 129 |
| Reordered Instruction Pair # <*proceeding inst., succeeding inst., reodering instruction distance (the # of instructions to be inserted between the proceeding and succeeding instructions*><br>†: “all” means all instructions in the instruction set. | | <sdza,all†,1><br><siza,all,1><br><reti,all,1><br><reta,all,1><br>:<br>Total = 13 pairs | <sdza,all,**2**><br><siza,all,**2**><br><reti,all,1><br><reta,all,1><br>:<br>Total = 15 pairs | <sdza,all,**3**><br><siza,all,**3**><br><reti,all,**2**><br><reta,all,**2**><br>:<br>(**swap,xorimm,1**)<br>**:**<br>Total = 315 pairs | same as the HT_4 case<br>(Total = 13 pairs) |

Figure 7. Architecture exploration for HT48100 embedded microcontroller