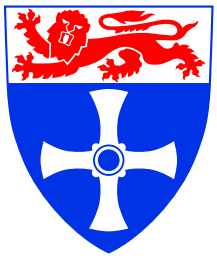
School of Electrical, Electronic & Computer Engineering

UNIVERSITY OF
NEWCASTLE UPON TYNE



# BUTLER Design and Analysis

Livin Koo

Technical Report Series

NCL-EECE-MSD-TR-2006-116

2006

Contact:

livinkoo@gmail.com

delong.shang@ncl.ac.uk

alex.yakovlev@ncl.ac.uk

NCL-EECE-MSD-TR-2006-116


School of Electrical, Electronic & Computer Engineering,
Merz Court,
University of Newcastle upon Tyne,
Newcastle upon Tyne, NE1 7RU, UK

http://async.org.uk/

# CONTENTS

***Abstract***

*Translating software scheduling functions into hardware has been extensively researched over the last decade. Different approaches and techniques like co-processor techniques, special purpose configurable hardware scheduler and customized hardware scheduler have been demonstrated for its improvement on system performance. In this paper, the focus will be put on the specification and documentation of a customized hardware scheduler design named BUTLER. The documentation presents the BUTLER in behavioural level and describes every function and search logic in details on top of the BUTLER design description by Eric Campbell [3]. Verilog specification is done for verification of the newly designed BUTLER with different scale and configuration.*

*As scheduling in embedded real-time multiple-processor systems being a major aspect in computer system resource management, the BUTLER plays a important role by handling most of the scheduling processes including interrupt control and context switching which maybe done by software functions in other hardware scheduling design. An important feature for the BUTLER is the flexible configuration, in which design tiles can configure in a different approach to obtain alternative parameters for a different design.*

*Keywords: hardware scheduler, interrupt control, context switching.*

***Acknowledgement***

I would like to take this opportunity to thanks my supervisor Professor Alex Yakovlev and Dr. Delong Shang. Valuable advices are given throughout the whole project period. Encouragement and useful suggestions on solving problems encountered during the project are the major elements of the success of the project. The kindly help from project topic selection to the solid work and dissertation preparation are all appreciated.

# 1. Introduction

Various studies have demonstrated that synchronous system operates well if the message exchange between the processors and memory modules of a multi-processor are of fixed length. However, lengths of messages are unlikely to be the same in reality, which makes asynchronous system becoming more efficient. As asynchronous systems going more and more important in modern computing systems, we would like to put our focus on the scheduling issues in embedded real-time multiple-processor systems. In order to achieve the highest efficiency in the destination processor, software functions tend to be replaced by some hardware solutions. Therefore, transferring task scheduling from software into hardware has been extensively investigated by researchers [1 – 3].

The design approaches and the level of hardware dependency on task scheduling are being compared among researches on hardware scheduler. Discussion of systems are categorised according to their own design approach, coarse grain, medium grain and fine grain. Since fabrication technology keeps improving, price of silicon drops. As a result, designs go to hardware that consumes more silicon area for higher circuit performance. Workload of CPU has to be minimized in order to maximize speed and performance in fast complicate multiple processes. On the other hand, the adoption of hardware schedulers to microprocessors is another problem addressed from recent researches.

## 1.1 Review of Literature

A co-processor design [1], a configurable hardware scheduler design [2] and a customized hardware scheduler design named BUTLER [3] represent exactly the three design approaches mentioned above. The following three sections are going to review all these approaches in both performance and design aspects.

### 1.1.1 Coarse grain

For hard systems which must provide very fast responses or support many application tasks or use complex, dynamic scheduling policies, real-time executive functions having minimal overheads are absolutely essential. A way to achieve this is to transfer

software functions into hardware by employing co-processing techniques – a 'software in silicon' solution. Such systems should limited to a small number of tasks to reduce task intercommunication time.

This type of task scheduler based on off-the-shelf hardware microcontroller. The approach is identifying all the tasks to be performed by the software and implementing as a set of co-operating processes. The overall system function is divided into a set of sub-functions or tasks and finally converted into sequential programs. The suggested task scheduler co-processor for hard real-time systems is particularly for reducing target system loading and in ease of interfacing [1].

#### 1.1.1.1 Scheduling issues

All activated tasks should be lying within one of the states shown in Fig. 1. Running is the state which a task is being executed, only one task can be in this state at any time. Ready is the state which task is waiting for access to the processor. And suspended is self-explanatory. For reasons, tasks are usually organised into queues as shown in Fig. 1. And the scheduler manages all the suspended queues including time and event management functions.

Schedule evaluation and task dispatching are the two operations involved in ready queue management. The former one determines how tasks should be ordered or prioritized. The latter one selects and installs the next application task required for execution by the processor.

#### 1.1.1.2 Performance

To achieve the best performance in processor of high utilization with minimization of missed deadlines, dynamic schedule is one of the solutions. However, with the high associated overhead when implemented in software, it seems to be ruled out from fast, hard systems. With a hardware scheduler, the associated overhead can be minimized for implementation. In return, the number of tasks should be kept to a minimum, so it is most suitable for single process.

In order to support different target processors, scheduler has to be language and processor independent, which makes impossible to prioritise tasks for the use of the

registers. All processor data must therefore be stored away during context switching, and normally placed off chip in RAM memory. Finally it becomes the major overheads in design involving co-processor.

#### 1.1.1.3 Functions

Scheduling and task switching decisions are all done by the co-processor with a specified scheduling policy. Besides, the target system must be able to avoid task switching, which may result in problems (e.g. a task being in a critical section of code). Furthermore, the co-processor was responsible for all task-timing functions like periodicity and delays. And task-level exception handling would be the duty of the unit as well, which centralized all error-handling decisions. It is intended to be an overall, centralized task controller for the whole system, which function with a selected scheduling policy. However, a certain number of scheduling policies may be available within the co-processor as well.

#### 1.1.1.4 Interfacing

Two kinds of connection related to the co-processor, interface between co-processor and target, and interface between co-processor and outside world. The former one is a single, simple processor independent connection. And the latter one including interrupt signals and serial data communication. In order to maintain full scheduling management and predictable operations, external interrupt signals are handled by the co-processor instead of the target processor. Turns out a more reliable system of higher security standard. The serial data communication is designed for the purposes of testing and performance evaluation only.

All system inputs except interrupts are all routed to the target and the same as all output signals for driving system are generated by the target processor, so the co-processor cannot directly change the outputs and thus modify system behaviour. Context switching is supported by interrupt signalling the target processor instead of direct access to the target by the co-processor unit.

### 1.1.2 Medium grain

To support high-resolution time tick in fast real-time applications with minimum overheads on the system, a configurable hardware scheduler for real-time systems is

proposed [2]. Its architecture reduced the time consuming in scheduling and time-tick processing to a minimum. The hardware scheduler is provided in the format of Intellectual Property (IP) blocks which allow designers to implement its own configuration with a developed tool. According to the scheduler suggested in the paper, three most common scheduling disciplines are supported, priority-based, rate monotonic and earliest deadline first [2]. It makes the scheduler more flexible and compatible to different target processors, where some of them might require scheduling policy changes during operation. Instead, some designs only focus on a specified scheduling policy which reduce the complexity on design but suffering from those systems need scheduling discipline changes.

The essence of the introduced system is the using of advance FPGA technology to implement part of the Real-Time Operating System (RTOS), in order to minimize scheduling and time-tick processing [2]. As the software scheduler and the time-tick processing are transformed into hardware component, the associated software overheads are eliminated as well. The configurable unit also overcomes the obstacle that hardware schedulers only supporting narrow range of applications faced by common hardware schedulers.

#### 1.1.2.1 Scheduling issues

Scheduling decisions are done by th e configurable hardware scheduler with the micro-architecture shown in Fig. 2. The operation is based on the information stored in Sleep queue, Priority queue, Task table, Current task registers and the control signals from the control unit. Moreover, its operation is associated with the interrupt signals from the interrupt controller as well.

Priority queue is a sorted queue used to store the active tasks according to its priority (ready queue). When a task is inserted, the queue automatically re-sorts itself in a priority order. Sleep queue is only responsible for storing the sleeping tasks (suspend queue). Task table is a look up table indexed by the task ID according to a specific task table entry format. Scheduler looks for task information from that unit whenever a task is activated. Interrupt controller is the unit responsible to handle all external interrupts, and pre-emption is supported as well. Finally, control unit acts as the

interface between the hardware scheduler and the external host, which receives and decodes commands and generates proper control signals to the system.

#### 1.1.2.2 Performance

For ordinary priority-based scheduler, the upper bound of the processing time is directly proportional to the number of tasks in the system. This overhead is large if there are many tasks and the time tick resolution is high. As a result, the CPU utilisation is reduced, and tasks may miss their deadlines. However, with the configurable hardware scheduler, assembly instructions executed by the scheduler and the background time tick processing are eliminated. Turns out the response time and the interrupt latency are improved [2].

#### 1.1.2.3 Functions

Time-tick (a periodic interrupt to keep track of time during which the scheduler makes a decision) handling, interrupt processing and execution of chosen scheduling algorithm are implemented in the hardware scheduler, while context switching is left to be done in software. Configuration of hardware and operations are performed by the software portion with a set of commands from the hardware scheduler. All commands are issued through a memory mapped I/O port, which can be done in one or two clock cycles depending on the size of the command word.

#### 1.1.2.4 Interfacing

When a task of higher priority is ready, the hardware scheduler directs the processor to task switching by sending a corresponding interrupt signal to the CPU. Upon receiving an interrupt signal, the CPU sends a control signal to the context switcher, which stores the current context, and switches to the task with the ID read from the hardware scheduler. The scheduler is designed in ease of interfacing with any microprocessor. The unit can either be a co-processor directly connected to the target processor or be a memory-mapped port connected to the system bus.

### 1.1.3 Fine grain

Eric Campbell (1996) introduced a physical device that can adapt to any microprocessor named BUTLER in the BUTLER design description [3]. The

BUTLER technology focused on the problems of scheduling application function tasks in embedded real-time multiple-processor systems, instead of the single-processor systems discussed in the previous sections. The device provides efficient support for multi-tasking in a single or multiple processor system. It holds all the control variables of tasks, which are going to execute in the microprocessor, and makes the decision for next running task during execution time. The device is particularly suitable for hard real-time embedded systems and systems that need to attain certain level of reliability.

#### 1.1.3.1 Scheduling issues

With reference to a programmed priority level selection, the BUTLER directly handles all asynchronous stimuli and schedules the relevant task when its turn arrives. Typical asynchronous stimuli are interrupt lines from local peripherals or from other connected BUTLERs. For all activities assigned to run on a processor, an associated BUTLER holds all control variables and computes the next activity that should be scheduled according to the current programmed priority levels and the control variable values. And instructions from the processor or asynchronous stimuli manipulate the control variables for scheduling purpose.

Eric Campbell {1996) suggested a priority leveling scheme that activities are numbered from zero to sixty-four, which named activity number. The smaller activity numbers the higher priority level [3]. Priority levels can be allocated to individual activities or to groups of activities. In case more than one activity is a candidate for scheduling at the same time, the next activity selection logic will select a candidate from the group with highest priority. Selection is made based on round robin if more than one candidate appears in the same priority group. Control interrupts will be generated under certain interrupt conditions.

#### 1.1.3.2 Performance

A design use asynchronous techniques throughout the whole system is presented. The BUTLER is totally responsible for asynchronous stimuli. It also claimed that the design could be easily implemented in different technologies because it is not dependent on critical timing parameters [3]. As a result, it avoids problems on clock signal distribution, clock skew or set-up and violation holding. Selection logic for

next activity is programmable. Furthermore, cooperative and pre-emptive scheduling schemes are also supported. The asynchronous design has non-demanding power supply requirements. However, the unit is not suitable for system environment with unintentional memory accesses as it performs specified functions when accessed as memory.

As mentioned before, multiple processor system is supported by the BUTLER hardware scheduler. Every single processor is connected to an associated BUTLER and BUTLERs between processor are connected with each other. Any scheduling request is done by registration with a processor's own BUTLER. BUTLERs communicate directly and schedule a relevant task to the destination processor during its turn if a task arrives on a different processor. This reduces the chance of interrupt the running task on the destination processor, and serving as a temporally deterministic operation.

#### 1.1.3.3 Functions

BUTLER can be used in conjunction with an associated processor hosting a small run-time software kernel. All BUTLER operations are performed in response to memory accesses from its associated local processor. The BUTLER directly handles all asynchronous stimuli, performs relevant task scheduling with reference to the programmed priority level selection. When accessed as memory, the BUTLER performs specified functions. Furthermore, the BUTLER also operates as an interrupt controller to handle all interrupts in scheduling process.

#### 1.1.3.4 Architecture and interfacing

The BUTLER has a tiling design approach – an assembly of design tiles in an array structure. Different types of design tile are butted together to form a two dimensional array without any additional signal routing between tiles. Each tile is a design building block that contains logic and structure, the array provides overall functions of the design.

For connection between BUTLER and its local processor, there is a standard memory interface consists of bi-directional data bus, address line inputs, memory control line inputs, interrupt output and a counter input line.

The main BUTLER array has eight different types of tiles, each built by a few simple gates. The main array is constructed by sixty-four rows of tiles each stores the control variables for an activity with a particular activity number incrementing from top row down the array. Without additional routing, a row of tiles that abut the top row of the main array forms interface to the processor, peripherals and other BUTLERs. Besides, array control signals are generated from that top row of tiles as well.

### 1.1.4 Conclusions

Numerous studies have demonstrated that hardware scheduler is an essential solution to reduce workload in CPU, improving efficiency and performance [1 – 3]. Different approaches of hardware scheduler implementations have been discussed in the above sections. Co-processor with microcontroller base, configurable hardware scheduler with FPGA base and the BUTLER fully customized scheduling hardware are compared for their performance and suitability for modern embedded real-time multi-processor system. Furthermore, the characteristics of scheduling, function area and interfacing problems of each design are addressed. It has been shown that programmability of circuit is decreased from unlimited programmable in coarse grain design to restricted programmable in medium grain design and finally no programming allowed in fine grain design. On the other hand, the performance of scheduling is increased from coarse grain to medium grain and achieves the highest performance in fine grain design.

The co-processor approach is a simple solution to translate the software functions into hardware; however, it only operates efficiently under single processor system which lies beyond our project focus area. The configurable hardware scheduler supports a wide range of processes and minimizes the scheduling of time-tick processing by implementing part of the RTOS in advance FPGA technology. The BUTLER design seems to be the best design for modern embedded real-time multiple-processor systems. It handles most of the scheduling tasks including interrupt control and context switching.

As a matter of fact that fine grain scheduling hardware is going to be our focus, this review therefore puts more effort on the BUTLER system’s scheduling issues, functions, architecture and interfacing. The design description claimed for its easy

implementation in different technologies by the independence on critical timing parameters. The asynchronous circuitry presented also shows no clocking or power supply requirement problems. An important feature for the BUTLER is the flexible configuration, in which design tiles can configure in a different approach to obtain alternative parameters. The investigation of different configurations is therefore part of the project area. Apart from those merits offered by the BUTLER system, the design being not conventional is a main weakness of the system. Therefore, a system documentation and a design specification including functionality are required for academics to follow.

## 1.2 Objectives

To improve microprocessor efficiency and performance, the aim of this project is to design a hardware scheduler based on the ad-hoc design of BUTLER. Since the original design is not conventional, a detail documentation of the BUTLER system is prepared and serves as reference for future studies and new system design in this project. Regarding the tiling design of the present BUTLER system, a new BUTLER with different tile configuration is designed to realize different functionality.

The original BUTLER able to handle 64 activities with different priorities, perform 16 different functions within itself initiated by memory access from microprocessor, handle external stimuli from local peripheral and other 4 connected BUTLERs. While the new designed BUTLER system handles 16 activities only, perform the same number of functions within BUTLER, number of connected BUTLER also reduced to one. In addition, size of the ripple down counter is reduced from 32 bits to 16 bits. Therefore, the total number of tiles in the main array may shrink from 1312 to less than 328.

Documentation consists of functional diagrams and description of functionality of the present ad-hoc BUTLER is produced to provide further information on top of the design description. The new BUTLER system design is presented in Verilog Hardware Description Language in Register Transfer Level for specification, while other simulation and analysis of the system is performed in the Cadence design environment. Functional block diagrams, system configuration and the final designed

system will be presented in later sections. Owing to the limited project period, implementation of the design will be a topic of future researches.

## 2. Methodology

Documentation of the original BUTLER system was produced according to the design description. In the following section, the documentation presents the detail of the BUTLER system in a behavioral level together with some major signal routings. Importance and realizations of different functions in the system are also demonstrated. In addition, system operations including various search logics are illustrated together with simplified block diagrams.

Several design tools made used during the design and simulation process are introduced briefly following the documentation. Procedures for the whole project are then discussed in the last part of the methodology section.

### 2.1 BUTLER Documentation

The BUTLER is a device provides efficient support for multi-tasking in a single or multiple processor system. It holds control variables for each task assigned to run on the microprocessor and continually identifies the next task that should run. Some control variables are manipulated by the instructions from the processor, some by asynchronous stimuli from local peripherals and some by asynchronous stimuli from other connected BUTLERs.

All BUTLER operations are carried out in response to memory accesses from its local processor. Write and read signals to the BUTLER are used to load operational data, return data to the processor and to initiate internal BUTLER operations. Different operations with respect to different addresses are shown in the table below. Request signals to initialize the operation will be sent out according to the indication of the stored instruction.

| Address | | | Write | Read |
|---|---|---|---|---|
| A2 | A1 | A0 | | |
| 0 | 0 | 0 | Load_Mask | Do_Stim |
| 0 | 0 | 1 | Load_Activity | Do_Wait |
| 0 | 1 | 0 | Do_Stimx | Suspend |

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 1 | Clear_All | Set_Suspended |
| 1 | 0 | 0 | Clear_Started | Set_Started |
| 1 | 0 | 1 | Clear_Pollend | Set_Pollend |
| 1 | 1 | 0 | Load_Counter_Lo | Nextact |
| 1 | 1 | 1 | Load_Counter_Hi | Control_Interrupts |

Table 1. BUTLER instruction addresses

### 2.1.1 Activity Priority

Activities are numbered from 0 to 64. When priority levels apply, smaller activity numbers have the higher priorities. Activity number 64 always has the lowest priority and can be used to schedule an idle activity at a time when no other activities are candidates for scheduling. Following a Clear_All BUTLER instruction activities numbered zero to sixty-four are assigned equal priorities. Priority levels can be allocated to individual activities or to groups of activities by inserting pollset boundaries, which is realized by Set_Pollend instructions. Pollset boundaries may be inserted or removed at any time.

### 2.1.2 System configuration

The overall system connection is shown in Fig. 3. Every BUTLER is connected with its own processor and four other BUTLERs. Four stimuli signals are connected to other four BUTLERs to select which BUTLER to stimulate and a six-bit address line is used to specify the activity number of a particular activity it wants to stimulate. Therefore, each BUTLER has 10 external stimuli output lines to other BUTLERs and 28 external stimuli input lines from other BUTLERs.

In Fig. 4, it shows the functional diagram of the present BUTLER. Decoded read, write and select signals together with a decoded address by three input address lines from the processor access the instruction memories to decide which instruction to operate. And then request signals are sent out from the instruction memories to initialize operation in other functional blocks as indicated in the figure. Most

functions involved in the next activity selection are performed within the activity memories. The next activity for scheduling is returned to the processor through the 16-bit bi-directional data bus (D) during a Nextact BUTLER instruction.

### 2.1.3 Instructions

**Load_Mask** (Write A000) is a BUTLER instruction to set the mask bit of different channels in the activity specified in the most recent Load_Activity instruction. The mask bits to set are specified in the data word (D) of a Load_Mask BUTLER instruction, where the data word is stored in the 16-bit bi-directional data bus. All sixteen bits of the data word are made used in this instruction to specify mask pattern for sixteen stim-wait channels. Some operations are performed to particular channels according to this mask bit.

**Load_Activity** (Write A001) BUTLER instruction sets the activity number that held on the BULTER. This activity number is specified in the data word (D) of a Load_Activity BUTLER instruction. Only the least significant six bits of the data word are used for this instruction to indicate activity number from zero to sixty-three. Some instructions operate on a specified activity with respect to this specified number.

**Do_Stim** (Read A000) is a BUTLER instruction to set the “Stimmed” channel according to the mask pattern specified in the most recent Load_Mask instruction of a particular activity specified in the most recent Load_Activity instruction. Those channels specified in that activity are considered to be stimmed when the “Stimmed” control variable is set, which is used to determine whether it is a candidate for scheduling in the next activity selection logic.

**Do_Stimx** (Write A010) BUTLER instruction generates external stimulation to other BUTLERs. Activity number specified in the most recent Load_Activity instruction is set up in the external stimuli output (Xout), which is connected to four other BUTLERs, in order to specify the activity to stimulate in the target BUTLER. Mask pattern specified in the most recent Load_Mask instruction is responsible for the selection of target BUTLER. As four other BUTLERs are connected to four different

channels respectively, the channel with mask bit set will generate a stimulation signal (Stimout) to the connected BUTLER.

**Set_Started** (Read A100) is a BUTLER instruction to set the “Started” variable of a particular activity true. The activity is chosen by the activity number specified in the most recent Load_Activity instruction. An activity is considered to be started when the “Started” control variable is set, which is used to determine whether it is a candidate for scheduling in the next activity selection logic. As there is only one “Started” control variable in each activity, no mask pattern is required during operation.

**Clear_Started** (Write A100) operates exactly the same way as Set_Started BUTLER instruction but to make false the “Started” variable instead of setting it true.

**Set_Pollend** (Read A101) is a BUTLER instruction to set the “Pollend” variable of the activity specified in the most recent Load_Activity BUTLER instruction. An activity with a set “Pollend” variable is served as the pollset boundaries for the next activity selection logic. Activities between two pollends are all assigned the same priority.

**Clear_Pollend** (Write A101) performs exactly the same way as Set_Pollend BUTLER instruction but to remove the pollset boundaries by resetting the “Pollend” variable instead of making it true. Pollset boundaries can be inserted or removed at any time by these two instructions.

**Load_Counter_Lo** (Write A110) BUTLER instruction loads the data word (D) into the least significant 16-bits of a 32-bit number, which is used to initialize the counter.

**Load_Counter_Hi** (Write A111) loads the data word (D) into the most significant 16-bits of the 32-bit number.

**Do_Wait** (Read A001) BUTLER instruction makes the “Waiting” variable true for the channels specified in the most recent Load_Mask BUTLER instruction for the activity currently running on the processor rather than the activity specified in the

Load_Activity BUTLER instruction. Those channels specified in that activity are considered to be waiting when the “Waiting” control variable is set, which is used to determine whether it is a candidate for scheduling in the next activity selection logic.

**Suspend** (Read A010) BUTLER instruction sets the “Suspended” variable true for the activity currently running on the processor. An activity is said to be suspended when the “Suspended” variable is made true, which is responsible to determine whether it is a candidate for scheduling in the next activity selection logic.
The Suspend or Do_Wait BUTLER instruction temporarily inhibits any changes to the asynchronously stimulated variables from entering the next activity selection logic.

**Set_Suspended** (Read A011) BUTLER instruction also sets the “Suspended” variable true but for the activity specified in the most recent Load_Activity BUTLER instruction instead of the activity currently running on the processor. As there is only one “Suspended” control variable in each activity, no mask pattern is needed during operation.

**Nextact** (Read A110) is the instruction used to return the number of the next activity that should be scheduled. The next activity to be scheduled is continually computed by the next activity selection logic based on the control variables held in BUTLER. All control variables are manipulated by local processor, local peripherals and other BUTLERs. The Nextact BUTLER instruction is also responsible to re-enable visibility of the asynchronous stimulated variables to the next activity selection logic, which is inhibited during context switch. The “Suspended”, “Waiting”, “Stimmed” variables of a particular activity will be reset when it is returned to the processor as the next activity to be scheduled.

**Clear_All** (Write A011) BUTLER instruction disables interrupts, removes any pollset boundaries (reset “Pollend” variables) and make false the “Started”, “Suspended” and all “Stimmed” and “Waiting” control variables for all activities.

**Asynchronous stimulations from four other BUTLERs** (Stimin) are connected to four different channels of the BUTLER. The stimulation will come along with its own address (Xin), which is the number of the target activity it wants to stimulate.

**Asynchronous stimulations from local peripherals** (Stimp) are made up of eight separate signals. Each signal is responsible for stimulating eight activities, eight signals for sixty-four activities. A particular channel in each activity is allocated to that asynchronous stimulation from local peripheral. Therefore, no activity number or mask pattern is required for the operation.
Both asynchronous stimulations are inhibited during context switch. That is invisible after Suspend or Do_wait BUTLER instruction and re-enabled visibility by Nextact instruction.

**Control_Interrupts** (Read A111) is a BUTLER instruction to enable the interrupt controller to function. Control interrupts are realized in two formats, pre-emption interrupt and counter interrupt. The instruction uses the least significant two bits of the activity number specified in the most recent Load_Activity BUTLER instruction (Actbit1 and Actbit0) to define its operation. All interrupts are reset during a context switch and are re-enabled after that. Following a Clear_All BUTLER instruction, the interrupt output line is reset to its non-active state.

### 2.1.4 Context Switch

A Suspend or Do_Wait instruction is used to inhibit visibility of any changes to the asynchronously stimulated variables from the next activity selection logic. Allowing enough time for the selection logic to stabilize before returning valid data to processor during the executing of Nextact instruction. The time period from a Suspend or Do_Wait instruction to a Nextact instruction is named context switch, when signal ‘slice’ will be taken high in the BUTLER internal circuitry.

### 2.1.5 Operation

**Decoder** is responsible for turning the values on input address lines, “Read”, “Write” and “Select” inputs into instruction request signals to initialize different instructions in other parts of the system. Three parts of decoding are involved in the decoder.

**Read-write signal decoding** simply passes the BUTLER I/O input Read to “R” and Write to “W” with an enabling input Select. Decoded signals “R” and “W” are then transmitted to instruction decoding to identify the instruction to trigger.

**Address decoding** is basically performing 3 to 8 decoding. Three input address lines are decoded to provide eight addresses for instruction decoding to identify instructions.

**Instruction decoding** produces appropriate request signals upon receiving signals “R”, “W” and the decoded address. Sixteen instructions are allocated as shown in Table 1. The request signals are routed to the functional blocks that responsible for the particular operation.

**Activity Number Register** stores the activity number specified in the more recent Load_Activity BUTLER instruction. When receiving request signal Load_Activity, the register latch in the least significant six bits of the bi-directional data bus (D) and stored as an activity number.

**Interrupt Controller** responses to handle two types of interrupts, pre-emption interrupt and counter interrupt. The least significant two bits of the activity number transmitted from the activity number register (Actbit1 and Actbit0) will decide which interrupt mode to perform. Operation of the interrupt controller is enabled by Control-Interrupts instruction request signal (Test) from instruction memories. When Actbit1 is high, Control_Interrupts is switched to pre-emption interrupt mode. If a candidate activity is found to have high priority than the one currently running on the processor, pre-emption interrupt is allowed to generate. The search is realized by a search logic that will be discussed in a later section. When Actbit1 is low, the system will prevent the generation of pre-emption interrupt.

When Actbit0 is high, Control_Interrupts is switched to counter interrupt mode. The interrupt output line will be set high upon receiving the signal from the counter to indicate the reaching of its limit. When Actbit0 is low, the system will prevent the generation of counter interrupt. Signal “Slice”, which indicating context switch, will inhibit any interrupts at its active state and enable further interrupts when it returns to

the non-active state. “Expired” from BUTLER counter will allow generating interrupt signal to the processor in the counter interrupt mode and “Maybe” from control memories search chain will further interrupt in the pre-emption interrupt mode.

**Counter**

The BUTLER has a 32-bit ripple down counter that counts low to high transitions on the counter input line (countin). A Do Wait or a Suspend BUTLER instruction initializes the counter to the 32-bit number that is held on the BUTLER. A Nextact BUTLER instruction enables the counter to start counting. The 32-bit number that is used to initialize the counter is programmable and is done by the Load_Counter_Lo and Load_Counter_Hi BUTLER instructions. A Load Counter Lo BUTLER instruction loads the data word into the least significant 16-bits of the number and a Load Counter Hi BUTLER instruction loads the data word into the most significant 16-bits of the number. When the counter receive a programmed number (plus one) of signal transitions on its counter input line, a signal “Expired” will be generated and sent to the interrupt controller for further interrupts.

**Mask Pattern Register** stores the mask pattern specified in the more recent Load_Mask BUTLER instruction. On receiving request signal Load_Mask, the register latch in data on the sixteen-bit bi-directional data bus (D) and stored as mask pattern. As the mask pattern information is only made used within the activity memories, the functional block is located inside the activity memories in the diagram.

**Control Memory** stores control variables for all sixty-four activities (Fig. 5) and perform instructions according to request signals from decoder. Performing instructions including Set_Started, Clear_Started, Set_Pollend, Clear_Pollend, Suspend, Set_Suspend, Do_Stim, Do_Stimx, Do_Wait, Nextact and Clear_All. Load_Mask BUTLER instruction is performed in the mask pattern register located inside the control memory functional block. One “Switch” latch for whole BUTLER generates signal “Slice” and distribute to rest of internal circuit during context switch.

### 2.1.6 Next Activity Selection

Based on the values of control variables it held, the BUTLER continually computes the next activity that should be scheduled, respecting the currently programmed priority levels. A Suspend or a Do_Wait BUTLER instruction temporarily inhibits any changes to the asynchronously stimulated variables from entering the next activity selection logic. A subsequent Nextact BUTLER instruction is used to return the number of the next activity that should be scheduled, and then to re-enable visibility of the asynchronously stimulated variables to the next activity selection logic. The logic will select an activity from the highest priority pollset that contains a candidate when more than one activity is a candidate for scheduling. Selection is made on a Round Robin basis within the pollset if this pollset contains more than one candidate. The search starts from the activity following the one that was last returned for scheduling in that pollset.

#### 2.1.6.1 Candidate for Scheduling

An activity will only be included as a candidate for scheduling when it is started and ready.

**“Started”**

An activity is started when its “started” control variable is true. A Set Started BUTLER instruction will make the “started’ control variable true for the activity specified in the most recent Load Activity BUTLER instruction. Vice versa, a Clear Started BUTLER instruction will set it false.

**“Suspended”, “Stimmed” and “Waiting”**

An activity is ready when either its “suspended” variable is true, or it has a matched pair of true “stimmed” and “waiting” variables. A Set Suspended BUTLER instruction will make the “suspended’ variable true for the activity specified in the most recent Load Activity BUTLER instruction. A Suspend BUTLER instruction will make the “suspended” variable true for the activity currently running on the processor (i.e. the last activity returned to the processor for scheduling). The “suspended” variable will be made false when the activity is returned to the processor as the next activity to be scheduled.

Each activity has sixteen pairs of “stimmed” and “waiting” variables, each pair is called a stim-wait channel. Some BUTLER instructions can operate on individual or groups of stim-wait channels. The stim-wait channels to be operated on are specified by including a one in an appropriate bit position in the data word of a Load_Mask BUTLER instruction. This mask pattern is held on the BUTLER.

A Do_Wait BUTLER instruction will make the “waiting” variable true for the stim-wait channel(s) specified in the most recent Load Mask BUTLER instruction for the activity currently running on the processor. The “waiting” variable will be made false when this activity is returned to the processor as the next activity to be scheduled.

A Do Stim BUTLER instruction will make the “stimmed” variable true for the stim-wait channel(s) specified in the most recent Load Mask BUTLER instruction for the activity specified in the most recent Load Activity BUTLER instruction. The “stimmed” variable will be made false when this activity is returned to the processor as the next activity to be scheduled.

External asynchronous “stimmed”
The “stimmed” variables on four particular stim-wait channels can also be asynchronously made true from an external source (e.g. by another BUTLER).
The “stimmed” variable on one particular stim-wait channel can also be asynchronously made true from a local peripheral.

#### 2.1.6.2 Polstart Search

Polstart search means the search of starting point of a Round Robin search logic within a pollset. Basically, the Round Robin search should be starting from the activity following the one last returned to the processor for scheduling. A “Last” latch indicates whether this is the last activity returned to the processor for scheduling, however, under certain circumstances more than one activity may indicate to be the last returned activity in this pollset. The logic will accept the first “Last” it found to be the polstart. In some cases, there will be no “Last” found for the whole pollset. The logic will set the activity with the smallest activity number in this pollset to be the polstart.

The search mechanism is shown in Fig 6. With the pollset boundaries set by the Set_Pollend BUTLER instruction, search logic chain is formed for each pollset. The search chain running down from the top of the pollset. Signal “Lastfnd” is set high when a “Last” is found in the search chain. An activity is considered as the polstart of this pollset if no “Last” has been found in the search and the “Last” latch of this activity is set. When the search encounters the pollend, the search is reset (i.e. reset signal Lastfnd) and starts again in the next activity. For the case “Lastfnd” is low when it encounters the pollend and the “Last” latch of this activity is not set (i.e. no “Last” has been found throughout the whole pollset), a signal “Lastloop” will be sent back to acknowledge the first activity in this pollset to be the polstart. Under this mechanism, exactly one “Last” latch set in each pollset can be achieved.

#### 2.1.6.3 Activities to be included in pollset of Nextact

When an activity is returned to the processor for scheduling, a latch “Here” is set. Two search chains are involved in this search for activities to be included in pollset of Nextact, one running up the activity array and another running down (Fig. 7). Both chains are taken low whenever they pass the pollset boundary and taken high when they pass the activity with “Here” set. Therefore, activities with a high in either search chain will be activities in the pollset of Nextact.

#### 2.1.6.4 Round Robin Implementation

When same priority is assigned to a group of activities, a Round Robin search is required to determine which activity to be returned next. Starting from the activity following the last activity returned to the processor, it runs through all activities in the pollset and loops back to the top of the pollset when it encounters the pollend. The search loop ends at the last activity returned to the processor (Fig. 8). When a candidate for scheduling is found in a pollset, signal “Search” will be taken low to indicate the rest of the pollset a candidate for scheduling has been found. An activity is selected to be the next activity if “Search” is high (no candidate has yet been found) and this activity is a candidate for scheduling. If a candidate has been found, signal “Found” is taken high at the end of the search loop and is passed to pollsets with

lower priority to indicate that candidate has already been found in higher priority pollset.

## 2.2 Design Tools

### 2.2.1 Verilog Hardware Description Language

A language used to describe digital systems at different levels, low implementation levels such as switch level and gate level; architectural or behavioral level like Register Transfer Level (RTL). In order to facilitate future studies on the BUTLER system, all Verilog specifications are done in register transfer level in this project.

### 2.2.2 Cadence Custom IC Design Tool

Cadence design system is particularly suitable for schematic generation, simulation, circuit synthesis and result analysis of digital or analogue circuitry. To verify the functions of different modules, all simulations are performed in the Cadence system design environment. Signal connections between different functional blocks are done in the schematic editor as symbol for every module is generated from the Verilog specification automatically.

### 2.2.3 Procedures

With reference to the BUTLER design description by Eric Campbell [3], block diagrams with different functional blocks are generated at early stage of the project. Together with the functional diagrams, behavioral logic of the system is derived in order to serve as a base of further specifications. Through understanding the original BUTLER design by Eric Campbell [3], documentation of the design is prepared for future studies including the design task in later part of the project. Different instructions, operation methodology and search logics are all introduced in the documentation.

Based on the functional diagrams and the documentation, Verilog specifications of the BUTLER system are firstly created in Verilog compiling software gVim. And the source files are then imported to the Cadence environment to test for the

functionalities by simulation. Stimulus files are programmed for the simulation under the Verilog XL simulator. Different functional modules are simulated independently with its own stimulus files. Overall system is simulated in the final stage when all functional modules are verified to function properly.

## 3. Results

System operating in the same manner with same functions of the original BUTLER is specified in the Verilog Hardware Description Language. However, compared with the original, the newly designed BUTLER is of a different scale as below:

- Number of activities is reduced from sixty-four to sixteen
- Number of stim-wait channels is reduced from sixteen to eight
- Counter size reduced from 32 bits to 16 bits
- Connected BUTLER reduced from four to one
- Local peripheral inputs reduced from eight to four
- Bi-directional BUTLER I/O data bus reduced from sixteen-bit to eight-bit

### 3.1 Verilog Specification

The specification is divided into five cells according to the functionality, decoder, activity number register, counter, interrupt controller and control memory. All cells operate asynchronously in response to request signals either from BUTLER I/O or other part of circuit.

#### 3.1.1 Decoder

As the number of instruction is the same as the original BUTLER, the structure of decoder is exactly the same as the original BUTLER. Read write signal decoding, address decoding and instruction decoding. BUTLER I/O “R” and “W” together with “select” trigger request signals “read” and “write” and which are transmitted to instruction decoding. Address input lines “a0”, “a1” and “a2” are decoded in the address decoding part, which is realized by a three to eight decoder. Sixteen instruction request signals are generated in the instruction decoding with reference to the decoded request signals (i.e. memory access from the local processor).

#### 3.1.2 Activity number register

The only difference in the activity number register is the size. It reduced from six bits for sixty-four activities to four bits for sixteen activities. Request signal “Load_Activity” from the decoder triggers the register to latch in data in the least significant four bits of the bi-directional BUTLER I/O data bus. This data are stored

as activity number by the most recent Load_Activity BUTLER instruction and transmitted to other functional blocks of the system.

### 3.1.3 Counter

As the scaling factor for the system is four, the counter size should be reduced to eight bits. However, this size may not be able to allow enough running time for all activities. A sixteen-bit counter is therefore chosen for the system. A Load_Counter_Lo BUTLER instruction triggers the counter to latch in the bi-directional data bus as least significant eight bits of a stored number, which is for the initialization of the counter. And a Load_Counter_Hi BUTLER instruction triggers the latching for the most significant eight bits. The counter is initialized by the stored sixteen-bit number on receiving signal “slice” during context switch. And it starts to count when “slice” goes low (i.e. after context switch). The counter counts the positive transition on the “countin” BUTLER I/O from the processor and generates a timeout signal “expired” when the count reaches the limit. The count starts from the number initialized by Load_Counter_Lo and Load_Counter_Hi BUTLER instruction. The timeout signal “expired” is reset by “slice” during context switch.

### 3.1.4 Interrupt controller

Two registers are used to indicate the mode of interrupt, “preemp_int” for preemption interrupt and “counter_int” for counter interrupt. With the “Test” signal from the decoder indicating interrupt control, the least significant two bits from the activity number register actbit1 and actbit0 decide the state of the registers. During interrupt control BUTLER instruction, “preemp_int” will be set when actbit1 is equal to one and reset when actbit1 is equal to zero; “counter_int” will be set when actbit0 is equal to one and reset when actbit0 is equal to zero. Signal “maybe” from control memory will enable interrupt if preemption interrupt mode is selected and “expired” from counter will enable interrupt if counter interrupt mode is selected. Both interrupt modes can be selected at the same time. Signal “maybe” and “expired” will be removed in the start of context switch. A Clear_All BUTLER instruction will reset both registers and disable interrupt.

### 3.1.5 Control memory

The main difference between the newly designed BUTLER and the original one is in the control memory. All control variables are stored in a control memory realized by twenty-one sixteen-bit registers. As shown in Fig. 9, each “stimmed” channel is specified by a sixteen-bit register, so as each “waiting” channel, “started”, “suspended”, “pollend”, “candidate” and “last” variables. Request signals Set_Started and Clear_Started set and reset the appropriate bit of the “started” variable respectively according to the activity number stored in the activity number register. Set_Pollend and Clear_Pollend set and reset the appropriate bit of the “pollend” variable respectively according to the stored activity number. Set_Suspended sets the appropriate bit of the “suspended” variables according to the stored activity number. Do_Stim sets the appropriate bit of the “stimmed” channels specified by the stored activity number and the stored mask pattern. The mask pattern loaded by the most recent Load_Mask BUTLER instruction will select the appropriate channels to operate.

Stimulation inputs from four local peripherals (stimp) are all connected to “stimmed” channel seven. Stimulations are made in an even distribution that each stimulation input is connected to four activities among sixteen. In order to allow enough time for the next activity selection logic to become stable, stimulation from local peripheral is disabled during context switch. Asynchronous stimulation can arrive at any time, however, the effect of the stimulation will be valid only after the context switch.
Asynchronous stimulation input from external source (i.e. from other connected BUTLER, stimin) is connected to “stimmed” channel six. Based on the activity number specified in the external address input lines (Xin[3:0]), “stimmed” variable of a particular activity is set. External asynchronous stimulation is inhibited during context switch, which means asynchronous stimulation arrived during context switch will be ignored.

Suspend and Do_Wait are instructions operating on the activity that currently running in the processor. As a matter of fact that activity number sixteen may be returned to the processor for an idle state when no candidate is found in the search, the instructions only operate on the “suspended” and “waiting” variables when it is not in

idle state (i.e. activity running in the processor does not has activity number sixteen). Suspend BUTLER instruction sets the “suspended” variable of the activity currently running in the processor. Do_Wait BUTLER instruction sets the “waiting” variables of the activity currently running in the processor with reference to the stored mask pattern.

On receiving a Load_Mask instruction signal from the decoder, data on the bi-directional data bus are latched into the mask register as the stored mask pattern for the operation of other instructions.

Overall reset of the system can be performed whenever request signal Clear_All from decoder arrives. All “stimmed”, “waiting”, “suspended”, “started”, “pollend” and “candidate” control variables will be reset during a Clear_All instruction and only the “last” variables are kept unchanged.

During Do_Stimx BUTLER instruction, the stored activity number will be transmitted to the external asynchronous stimulation address output lines (“Xout [3:0]”). Stimulus (“stimout”) will be set and transmitted to other connected BUTLER together with the target activity number on “Xout”.

Signal “slice” is passed around to indicate during context switch. It can be made true by either Suspend or Do_Wait BUTLER instructions and reset by a Nextact.

During Nextact BUTLER instruction, activity output (“act_out”) from the next activity selection logic will be passed to the bi-directional data bus and stored in a register (“act_run”) as well. If the returning activity is not idle, all “stimmed”, “waiting”, “suspended” and “candidate” control variables of the returning activity will be reset. In this case, a search logic is used to reset the “last” variable of all other activities within the same pollset of the returning activity and the “last” variable of it will be set.

Fig. 10 shows a clearer picture of how the search logic runs. Firstly, the “last” variable of the returning activity is set. Secondly, a search chain is running down every activity from the one following the returning activity. The “last” variable of all

activities passed by the chain is reset until a pollset boundary is found. When a set "pollend" variable is found, variable "pollend_run" is set to indicate the end of search. Thirdly, a search chain is running up from the activity above the returning one. The "last" variable of all activities passed by the chain is reset until a pollset boundary is found. When a set "pollend" variable is found, variable "polltop_run" is set to indicate the end of search. Finally, in case of the returning activity is either zero or fifteen, only search running down or running up will be performed respectively. The search logic will not be initialized if the returning activity is activity sixteen.

As the BUTLER computes the next activity to be scheduled continually, searches related to the next activity selection logic perform continually as well. A loop running from activity zero to activity fifteen is used to set the "candidate" variable when an activity is ready and started. An activity is ready means either the "suspended" variable or a matched pair of "stimmed" and "waiting" variable is set. Started means the "started" variable is set.

### 3.1.6 Searches

#### 3.1.6.1 Next activity selection

First step of the next activity search is to find out set "last" variable from activity zero to activity fifteen. When a set "last" variable is found, a Round Robin search will be performed. The specification of the Round Robin search is divided into three groups, one for "last" variable of activity zero, one for "last" variable of activity fifteen and one for other activities. As precisely one set "last" variable in each pollset is guaranteed by another search logic in the system, each pollset will be ran by exactly one Round Robin search. Search will be started from the pollset with highest priority.

**Round Robin search for set "last" variable in between activity one and activity fourteen**

If "pollend" variable of that activity with set "last" is low, a search going down the array is started. Starting from the activity following the one with set "last", variable "searchend" will be set when a set "pollend" variable is found. Down search will be stopped and an up search will be followed. Starting from the activity with set "last", variable "searchtop" will be set when a set "pollend" variable is found in the up

search. Search for this pollset will be stopped and another Round Robin for the next pollset will be followed. When a set “candidate” variable is found in the search, activity number will be passed to the search output (“act_out”) and variable “found” will be set. As a matter of priority in a Round Robin search, variable “found” can stop the down search but not the up search. In down search, searching process finishes when candidate is found. In up search, searching process runs until pollset boundary. Candidates found in the later part of up search can overwrite the search output in earlier part of the same up search. Search in other pollset will not be started if variable “found” is set to indicate a schedulable activity has been found. If “pollend” variable of that activity with set “last” is high, only up search will be performed in the pollset.

**Round Robin search for activity zero being “last”**

If “pollend” variable is not set, a down search will be started and followed by checking the “candidate” variable of activity zero itself. If “pollend” variable of activity zero is set, “candidate” variable will be checked and no search is performed within the pollset.

**Round Robin search for activity fifteen being “last”**

Only up search will be performed until set “pollend” variable is found in the search.

If “found” equals to zero indicating no candidate has been found after searching through all fifteen activities, activity sixteen will be returned to the processor for idle state.

#### 3.1.6.2 Precisely one set “last” variable in each pollset guarantee

This logic involves two parts, one to remove extra set “last” variable in a pollset and one to set “last” variable when no “last” found in a pollset. To remove extra set “last” variable, a search chain running down from top of array is used. Variable “lastfnd” is set high when the chain encounters a set “last” variable and is reset when it encounters a set “pollend” variable (i.e. “lastfnd” reset in each pollset). All “last” variables are reset by the chain when “lastfnd” is high.

To set the activity with smallest activity number in the pollset to be “last” when no set “last” has been found, a search chain running up from the bottom of array is used.

Variable “lastfndup” is set high when the chain encounters a set “last” variable and reset when it encounters a set “pollend” variable like “lastfnd” in the down running chain. When the chain encounters a set “pollend”, activity in top of the pollset will be set if “lastfndup” is low indicating no set “last” has been found in the pollset. In top of the array, there is no “pollend” above activity zero to indicate a pollset boundary. Therefore, “last” variable of activity zero is set by the chain if “lastfndup” is low. If “pollend” variable of activity zero is set, the “last” variable will also be set as it is the only activity in the pollset.

#### 3.1.6.3 Maybe search

This is the search for candidate with higher priority than the one currently running in the processor. Basically, this search should run continuously as long as the system is powered up. However, base on the fact that a candidate with higher priority can only exist when there is a change in the output of the next activity selection logic (“act_out”), this “maybe search” will only perform when the value of “act_out” changes. As interrupt is inhibited during context switch, this search is suspended during context switch.

The search starts from the activity currently running in the processor but the checking for “candidate” would not be started until a set “pollend” variable is found in the search. When a “pollend” variable is found, variable “search_start” will be set high and indicating the start of checking “candidate” variable. When a set “candidate” variable is found, signal “maybe” is set high and transmitted to the interrupt controller for the generation of interrupt. Signal “maybe” is reset by “slice” during context switch.

## 3.2 Verification

Each cell is simulated independently to verify it function properly and an integrated simulation is done to verify the system operation. Decoder, activity number register, counter, interrupt controller and control memory are simulated independently before integration to reduce complexity on the final integrated simulation. Two versions of control memory are simulated with similar stimulus files, where basic control variable

setting functions are verified in the simple version first and then the precise search logic is verified in the final version. As basic operations are verified in the simple version, the simulation for the final version is much less complicated but to concentrate on the BUTLER customized next activity search logic.

As BUTLER is designed for asynchronous operations, the next activity selection logic operates continuously. In the original design, the outputs from the logic are put onto the bi-directional data bus during all BUTLER read instructions; in the new behavioral specification, outputs are put onto the data bus during Nextact BUTLER instruction only.

### 3.2.1 Decoder

Sixteen different request signals are simply simulated by reading from and writing to all combinations of input address lines. Output signals from the decoder are monitored. Signal waveform is shown in Fig. 11. Request signals are triggered in turn by read, write, select signals and input address lines a0, a1 and a2.

### 3.2.2 Activity number register

This four-bit register is simulated by different values on the bi-directional data bus together with Load_Activity enable signal. Latched activity number is monitored. Signal waveform is shown in Fig. 12.

### 3.2.3 Counter

Different values on the eight-bit bi-directional data bus are loaded into the sixteen-bit internal register (“stored”) of the counter by Load_Counter_Lo and Load_Counter_Hi input. After the counter is initialized by signal “slice”, initial count value is monitored to verify the Load_Counter_Lo and Load_Counter_Hi functions. Periodic signal “countin” with period of two time units is kept feeding to the counter until the count reaches the limit and set “expired” high. Simulation is run until “expired” is reset by the following context switch. A test output “countout” is used to monitor the count value throughout the whole simulation. Signal waveform is shown in Fig. 13.

### 3.2.4 Interrupt controller

Preemption interrupt and counter interrupt are simulated with signal input “maybe” and “expired” with “slice” keeps low at the beginning. Output “interrupt” is monitored. Input “slice” is then altered to check if interrupt is inhibited during context switch. Finally, the least significant two bits of the activity number inputs are reset in turn to check if both interrupt mode can be disabled by setting corresponding activity bit to zero. Signal waveform is shown in Fig. 14.

### 3.2.5 Control memory (simple version)

As the search logic in this version is only a top down search (i.e. search from activity zero to activity fifteen), all functions involving the search logic is not simulated in this version.

Activity number is provided at the input to simulate the connection with activity number register. A Load_Mask instruction is performed first to provide mask pattern for operation in later part of simulation, which is realized by a Load_Mask request signal and data in the eight-bit bi-directional data bus. Data is stored in the register “mask” to serve as a stored mask pattern. Basic control variable setting operations Set_Started, Clear_Started, Set_Pollend, Clear_Pollend and Set_Suspended are simulated with activity number input. “Started”, “pollend” and “suspended” variable registers are connected to the test outputs (“testout”) and the test output is monitored to verify all the functions mentioned above. BUTLER instruction Do_Stim is then simulated with activity number input and the stored mask pattern. “Stimmed” channels six and seven are connected to the test output to verify function Do_Stim and the correctness of function Load_Mask.

Stimulations from local peripherals (“stimp”) are scheduled to arrive both during context switch and between context switches. “Stimmed” channel seven is connected to the test output to monitor the stimulation. Effect of stimulus arrive during context switch should be deferred till the end of context switch.

When instruction Suspend is being simulated, “suspended” variable register is connected to the test output. “Suspended” variable of the activity currently running in the processor is set if the processor is not running activity sixteen (idle state). Signal

“slice” should be high to indicate a context switch. Instruction Nextact is followed to complete a normal context switch, “slice” returns to low. Since there is no activity running in the processing during start up, the above context switch process is performed twice to obtain a correct result.

A Clear_All BUTLER instruction is performed once at the beginning to ensure all control variables are in their non-active state and it is test after a context switch cycle. All control variables are monitored to verify the correctness of a Clear_All instruction.

To verify the Do_Stimx function, output “Xout” is monitored to check if activity number from input “act_no” is transmitted to output “Xout” when Do_Stimx is high. Asynchronous stimulation from external source is simulated by specifying an activity number in the input “Xin” and “stimmed” channel six is monitored. Signal waveform in Fig. 15 shows all the operation mentioned above accordingly.

### 3.2.6 Control memory (final version)

Following a Clear_All BUTLER instruction, “last” variable register is monitored to check if any “last” variable is set by the one “last” guarantee logic. Activities four, five and six are then “started” and “suspended” to make them candidates for scheduling, and the “candidate” variable register is monitored. A context switch is performed to return activity from the search. Pollset boundaries are set up to form priority groups. Activity two and fourteen are set to be pollset boundaries. Round Robin search is verified by allowing three candidates in the same pollset and running context switch several times. Activity from pollset with higher priority (activity zero) is made to be candidate to test the “maybe search” logic. Clear_Pollend instruction is performed to make more than one set “last” in a pollset and “last” variable register is monitored to check the one “last” guarantee logic. Finally, a context switch is performed after a Clear_All instruction to check if activity sixteen is returned by the search logic when no candidate is found. Signal waveform is shown in Fig. 16.

### 3.2.7 Integrated simulation

A final simulation is performed to verify the operation of the whole system, which is connected as shown in Fig. 17. Stimulus file used is of the same logic as the one used

in the control memory, together with the stimulus of other functional modules. Stimulus inputs become BUTLER I/Os instead of internal signal routings. Operations are initialized by memory access and stimulation from external sources. BUTLER I/Os are monitored to verify correct operation. Signal waveform is shown in Fig. 18.

## 4. Analysis and discussion

To improve microprocessor efficiency and performance by replacing software-scheduling program by a hardware solution, a hardware scheduler is designed with reference to the ad-hoc BUTLER design. In order to facilitate future research, a documentation of the BUTLER design is prepared on top of the BUTLER design description by Eric Campbell [3].

### 4.1 Operation by memory access

The BUTLER performs specified functions when accessed as memory. It should not therefore be used where unintentional memory accesses may occur, such as in direct-memory-access, cache or refresh memory systems.

### 4.2 Round Robin

A round robin is an arrangement of choosing all elements in a group equally in some rational order, usually from the top to the bottom of a list and then starting again at the top of the list and so on. A simple way to think of round robin is that it is about "taking turns". In computer operation, different program processes take turns using the resources of the processor is to limit each process to a certain short time period, then suspending that process to give another process a turn (or "time-slice"). This is often described as Round Robin process scheduling. In this project, Round Robin search is made use in next activity selection logic when more than one activity are eligible for scheduling within the same priority group.

### 4.3 Counter

Arrays of flip flops which have the property of incrementing or decrementing when pulsed are known as counter registers, or counters. Normally each bit of the binary code is stored in a flip flop, with N flip flops giving up to $2^N$ states. Essentially there are two kinds of counters. Synchronous counters have all flip flops simultaneously clocked by the count pulse. In asynchronous circuits, normally only the first flip-flop is directly clocked; this change is then propagating through the remaining logic.

The main advantage of ripple counter is its relative simplicity. However, their asynchronous nature gives problems in some situations. Because of cumulative delays as changes propagate along the chain, some alterations of state occur in a staggered manner. Counter delay is not a significant problem in this system as counter is only used for control interrupt.

## 4.4 Memory VS Registers

Control variables of all activities are stored in a control memory, which can be accessed by different BUTLER instructions and asynchronous stimulations. In the original BUTLER design, the memory is realized by a tile configuration. In the behavioral specification, a decision between memory and registers has to be made to specify the control memory.

As memory cannot be referenced at the bit-level in Verilog HDL, data in the word have to be first transferred to a temporary register. Therefore, temporary registers have to be made used throughout the whole specification during specific bit range operation. In this case, concurrent operation of same word maybe problematic, which means asynchronous stimulation may not be supported. Therefore, control variables should be held by twenty-one registers instead of a sixteen by twenty-one memory. Asynchronous stimulation can arrive any time concurrently with different BUTLER instructions.

## 4.5 Arbitration Problem

Since BUTLER deals with different asynchronous operations, set and reset of certain control variable latches may occur during normal operation. Additional circuitries are added to avoid any erroneous state resulted from this.

### 4.5.1 Last Latch

For normal operation, exactly one “last” latch should be set within each pollset. However, none of the “last” latch will be set in the whole control variable memory during the initial power up. As pollset boundaries can be set or removed at any time,

no set “last” latch or more than one “last” latch in one pollset may occur during Set_Pollend, Clear_Pollend or Clear_All BUTLER instructions. Additional circuitries are therefore designed to maintain normal operation. Since no pollset boundary is set during initial power up, one pollset contains all activities. “last” latch of the zeroth activity, top of the pollset, will be set to retain normal operation. For the reset of pollset boundaries, activity with the smallest activity number in the pollset will be set if no “last” latch found. And the search chain will only take the first set “last” latch it found and ignore the others if multiple set “last” latches have been found.

### 4.5.2 Asynchronous Stimulation

During execution of Clear_All or Nextact BUTLER instruction, some or all “stimmed” latches will be initialized (make false). However, asynchronous stimulation from external source or local peripherals may arrive at any time, which will make true the “stimmed” latch of some channels according to data carried. Concurrent set and reset of the “stimmed” latch can occur when stimin from an asynchronous external source is concurrent with reset from BUTLER instruction Clear_All or Nextact when this activity is being returned to the processor. The normally complementary outputs from the “stimmed” latch will both be high. This causes no problem because this activity will be being returned to the processor as the next activity to be scheduled at this time. If removal of concurrent set and reset are coincident, the “stimmed” latch will, after the delay needed to resolve the metastability effect, become either set or reset. Time is available between executing BUTLER instructions for the latch to settle. If it becomes set, “stimin” is assumed to have occurred after Nextact; if it becomes reset, “stimin” is assumed to have occurred before Nextact. Either condition provides correct system operation.

To avoid arbitration problem, additional circuitry is added to defer visibility of a “stimmed” latch that is set by asynchronous local peripheral (“stimp”) during context switch. Extra latch is added to store value of the “stimmed” latch until after a subsequent Nextact BUTLER instruction.

## 4.6 Implications and Practical Applications

Base on the BUTLER documentation and the Verilog specification, BUTLER of different functions and scale can be easily designed in the future. As two versions of BUTLER with different search logic are specified in this project, different search logic can also be used in the future, by simply replacing the search logic section in the present specification. In practical, BUTLER can be used from complex system like a computer microprocessor to simple system like processor in toy. It can improve the efficiency and performance of the processors in both systems.

## 5. Conclusion

As a matter of fact that asynchronous designs are getting more important in the state-of-art computer system designs, modern computing systems tend to move synchronous design to asynchronous. To improve efficiency and performance of processors, scheduling functions are moved from software to hardware. From the comparison made in the literature review section, advantages of different hardware dependency level of scheduler are presented. The BUTLER technology is focused on the issues of scheduling application function tasks in embedded real time multiple processor systems. In this project, a detail documentation of the original design has been prepared to provide sufficient information for future studies and implementations. Different functions, operations, precise search logics, signal routings and control variables for next activity selection are all presented in details. This documentation can support future design in a behavioral level on top of the gate level design description by Eric Campbell [3].

The documentation described the BUTLER from general functions to detail operation logic, from system configuration to BUTLER internal signal routings. It explained detail of the next activity selection logic, the system configuration of the original design, actual function of different tasks, operation of different function blocks in the system and all search logics help maintain correct operation of system. Functional diagram and search logic diagrams are included to illustrate some complicate search logics and major signal routings.

Besides the documentation, the description of Verilog specification presented in the result section verifies the functionality of the system and the possibility of varying the configuration of the original design. Although the newly designed BUTLER in this project is specified in register transfer level, a synthesis can be performed by following the tiling approach of the original design. Since the new design is of different scale and configuration with the original, it verified the possibility of different number of activities and connected BUTLERs.

As the Verilog specification of the BUTLER is divided into five modules according to the functional block diagram presented in the documentation, each module is presented independently for its function and operation logic. Functions triggered by

different request signals from external sources or internal circuitry are explained in full details. Search loops that run continuously throughout the whole operation of BUTLER are presented with the aid of block diagrams to enhance understanding of those complicate search logics.

A verification section is followed to display the precise simulation procedure carried out in the project. Stimulus file used in the verification of each module is described independently, followed by the description of an integrated simulation for the whole system. Simulation results proved the correctness of the new design and therefore, the possibility of changing scale and configuration of the original design. Future studies should focus on the synthesis in the tiling approach and implementation of different configuration BUTLER. The documentation provided a channel for academics to understand the BUTLER operation in an efficient way and the Verilog specification provided a basic design methodology of the BUTLER for researchers to follow.

**Reference**

1. J. E. COOLING, P. TWEEDALE**,** "Task scheduler co-processor for hard real-time systems", Department of Electronic and Electrical Engineering, Loughborough University of Technology, Loughborough, Leicestershire, UK. 20 December 1996.

2. PRAMOTE KUACHAROEN, MOHAMED A. SHALAN and VINCENT J. MOONEY III, "A Configurable Hardware Scheduler for Real-Time Systems", Centre for Research on Embedded Systems and Technology, School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, Georgia 30332, USA.

3. ERIC CAMPBELL, "BUTLER design description", Computing Systems Technology Department, British Aerospace Defence Ltd (Dynamics Division), DR11481, Issue 1, April 1996.

## Appendix

Appendix 1

**----------Counter ----------**

```
module counter(expired, countout, Test, slice, D, countin, Ld_cntr_lo, Ld_cntr_hi);
input Test, slice, countin, Ld_cntr_lo, Ld_cntr_hi;
input [7:0] D;
output expired;
output [15:0] countout;
reg [15:0] stored, count;
reg time_out;
//
initial begin
        stored = 16'b0;
        count = 16'b0;
        time_out = 0;
end
//
always @ (posedge countin)
begin
        if (~slice)
                count = count + 1;
        if (count == 16'b0)
                time_out = 1;
end
//
assign expired = time_out;
//
always @ (Ld_cntr_lo)
begin
        if (Ld_cntr_lo)
                stored[7:0] = D[7:0];
end
//
always @ (Ld_cntr_hi)
begin
        if (Ld_cntr_hi)
                stored[15:8] = D[7:0];
end
//
always @ (slice)
begin
        if (slice)
        begin
                time_out = 0;
                count = stored;
        end
end

assign countout = count;
endmodule
```

**----------Interrupt controller ----------**

```
// Created by ihdl
module int_cntl(interrupt, act_no, Test, expired, maybe, slice, Clrall);
input [3:0]act_no;
input Test, expired, maybe, slice, Clrall;
output interrupt;
reg counter_int, preemp_int;
```

```
//
initial begin
	counter_int = 0;
	preemp_int = 0;
end
//
always @ (Test)
begin
	if (Test == 1 & act_no[1] == 1)
		preemp_int = 1;
	if (Test == 1 & act_no[1] == 0)
		preemp_int = 0;
	if (Test == 1 & act_no[0] == 1)
		counter_int = 1;
	if (Test == 1 & act_no[0] == 0)
		counter_int = 0;
end
//
always @ (Clrall)
begin
	if (Clrall == 1)
	begin
		preemp_int = 0;
		counter_int = 0;
	end
end
//
always @ (act_no)
begin
	if (Test == 1 & act_no[1] == 1)
		preemp_int = 1;
	if (Test == 1 & act_no[1] == 0)
		preemp_int = 0;
	if (Test == 1 & act_no[0] == 1)
		counter_int = 1;
	if (Test == 1 & act_no[0] == 0)
		counter_int = 0;
end
//
assign interrupt = (preemp_int & maybe & ~slice) | (counter_int & expired & ~slice);
endmodule


----------Activity number register ----------
module act_no_reg(act_no, Ld_act, D);
input [7:0]D;
input Ld_act;
output [3:0]act_no;
reg [3:0]act_reg;

initial begin
	act_reg = 0;
end

always @ (Ld_act)
begin
	if (Ld_act)
		act_reg[3:0] <= D[3:0];
end
	assign act_no[3:0] = act_reg[3:0];
```

```
endmodule
```

**Decoder**

```
module add_dec(a000, a001, a010, a011, a100, a101, a110, a111, a0, a1, a2);
input a0, a1, a2;
output a000, a001, a010, a011, a100, a101, a110, a111;
//
assign a000 = ~a0 & ~a1 & ~a2;
assign a001 = ~a0 & ~a1 & a2;
assign a010 = ~a0 & a1 & ~a2;
assign a011 = ~a0 & a1 & a2;
assign a100 = a0 & ~a1 & ~a2;
assign a101 = a0 & ~a1 & a2;
assign a110 = a0 & a1 & ~a2;
assign a111 = a0 & a1 & a2;
//
Endmodule

module rw_decoder(read, write, R, W, select);
input R, W, select;
output read, write;

assign read = R & select;
assign write = W & select;

endmodule

module instr_memory(Do_stim, Do_wait, Suspend, Setsus, Sstart, Spollend, Nextact, Test, Ld_mask,
Ld_act, Do_stimx, Clrall, Cstart, Cpollend, Ld_cntr_lo, Ld_cntr_hi, a000, a001, a010, a011, a100,
a101, a110, a111, read, write);
input a000, a001, a010, a011, a100, a101, a110, a111, read, write;
output Do_stim, Do_wait, Suspend, Setsus, Sstart, Spollend, Nextact, Test, Ld_mask, Ld_act,
Do_stimx, Clrall, Cstart, Cpollend, Ld_cntr_lo, Ld_cntr_hi;
//
assign Do_stim = read & a000;
assign Do_wait = read & a001;
assign Suspend = read & a010;
assign Setsus = read & a011;
assign Sstart = read & a100;
assign Spollend = read & a101;
assign Nextact = read & a110;
assign Test = read & a111;
assign Ld_mask = write & a000;
assign Ld_act = write & a001;
assign Do_stimx = write & a010;
assign Clrall = write & a011;
assign Cstart = write & a100;
assign Cpollend = write & a101;
assign Ld_cntr_lo = write & a110;
assign Ld_cntr_hi = write & a111;
//
endmodule
```

**----------Conrol Memory (simple version) ----------**

```
module Cntl_Mem01(Xout, Dout, stimout, maybe, slice_out, testout0, testout1, testout2, testout3,
testout4, testout5, testout6, Xin, stimin, D, Sstart, Cstart, Spollend, Cpollend, Setsus, Do_stim,
Do_stimx, Nextact, Do_wait, Suspend, Clrall, Ld_mask, act_no, stimp0, stimp1, stimp2, stimp3);
output stimout, maybe, slice_out;
output [3:0] Xout;
output [4:0] Dout;
```

```
output [0:15] testout0, testout1, testout2, testout3, testout4, testout5, testout6;
input [3:0] act_no, Xin;
input [7:0] D;
//BUTLER instructions request signals from instruction memory
input Sstart, Cstart, Spollend, Cpollend, Setsus, Do_stim, Do_stimx, Nextact, Do_wait, Suspend,
Clrall, Ld_mask, stimin;
//stimulation from local peripherals
input stimp0, stimp1, stimp2, stimp3;
//16 activities, 8 pairs of stim-wait channels [0-7 stim, 8-15 wait]
//Suspend[16], Start[17], Pollend[18], Candidate[19], Last[20]
reg [0:15] stim_bit0, stim_bit1, stim_bit2, stim_bit3, stim_bit4, stim_bit5, stim_bit6, stim_bit7,
wait_bit0, wait_bit1, wait_bit2, wait_bit3, wait_bit4, wait_bit5, wait_bit6, wait_bit7;
reg [0:7] mask;
reg [3:0] act_x;
integer i, j, k, m, n, p, act_run, act_out;
reg slice, Lastfnd, Lastfndup, candfnd, maybe_reg, pollend_run, polltop_run;
reg [0:15] suspend_bit, start_bit, pollend_bit, cand_bit, last_bit;
//
initial begin
act_x = 4'bzzzz;
slice = 0;
Lastfnd = 0;
Lastfndup = 0;
candfnd = 0;
maybe_reg = 0;
pollend_run = 0;
polltop_run = 0;
end
//
/////////Set Start, Clear Start, Set Pollend, Clear Pollend, Set Suspend, Do Stim.////////
/////////Instruction act on activity specified in last Load Activity instruction.////////
//
always @ (Sstart)
begin
        if (Sstart)                                     //set start
                start_bit[act_no] = 1;
end
//
always @ (Cstart)
begin
        if (Cstart)                                     //clear start
                start_bit[act_no] = 0;
end
//
always @ (Spollend)
begin
        if (Spollend)                                   //set pollend
                pollend_bit[act_no] = 1;
end
//
always @ (Cpollend)
begin
        if (Cpollend)
                pollend_bit[act_no] = 0;
end
//
always @ (Setsus)
begin
        if (Setsus)                                     //set suspend
                suspend_bit[act_no] = 1;
```

```
end
//
always @ (Do_stim)
begin
	if (Do_stim)					//set stim according to mask
	begin
		if (mask[0] == 1)
			stim_bit0[act_no] = 1;
		if (mask[1] == 1)
			stim_bit1[act_no] = 1;
		if (mask[2] == 1)
			stim_bit2[act_no] = 1;
		if (mask[3] == 1)
			stim_bit3[act_no] = 1;
		if (mask[4] == 1)
			stim_bit4[act_no] = 1;
		if (mask[5] == 1)
			stim_bit5[act_no] = 1;
		if (mask[6] == 1)
			stim_bit6[act_no] = 1;
		if (mask[7] == 1)
			stim_bit7[act_no] = 1;
	end
end
//
////////////////////////////Set stim by local peripherals////////////////////////////////
always @ (posedge stimp0)
begin
	if (stimp0)
	begin
		wait (~slice)
		begin
			stim_bit7[0] = 1;
			stim_bit7[4] = 1;
			stim_bit7[8] = 1;
			stim_bit7[12] = 1;
		end
	end
end
//
always @ (posedge stimp1)
begin
	if (stimp1)
	begin
		wait (~slice)
		begin
			stim_bit7[1] = 1;
			stim_bit7[5] = 1;
			stim_bit7[9] = 1;
			stim_bit7[13] = 1;
		end
	end
end
//
always @ (posedge stimp2)
begin
	if (stimp2)
	begin
		wait (~slice)
		begin
```

```
                        stim_bit7[2] = 1;
                        stim_bit7[6] = 1;
                        stim_bit7[10] = 1;
                        stim_bit7[14] = 1;
                end
        end
end
//
always @ (posedge stimp3)
begin
        if (stimp3)
        begin
                wait (~slice)
                begin
                        stim_bit7[3] = 1;
                        stim_bit7[7] = 1;
                        stim_bit7[11] = 1;
                        stim_bit7[15] = 1;
                end
        end
end
//
//
////////////////////////////Suspend, Do Wait.////////////////////////////////////
//////////instruction act on activity currently running in the processor.//////////
//
always @ (Suspend)
begin
        if (Suspend)
        begin
                if (act_run != 16)
                        suspend_bit[act_run] = 1;
        end
end
//
always @ (Do_wait)
begin
        if (Do_wait)                                    //set wait according to mask.
        begin
                if (act_run != 16)
                begin
                        if (mask[0] == 1)
                                wait_bit0[act_no] = 1;
                        if (mask[1] == 1)
                                wait_bit1[act_no] = 1;
                        if (mask[2] == 1)
                                wait_bit2[act_no] = 1;
                        if (mask[3] == 1)
                                wait_bit3[act_no] = 1;
                        if (mask[4] == 1)
                                wait_bit4[act_no] = 1;
                        if (mask[5] == 1)
                                wait_bit5[act_no] = 1;
                        if (mask[6] == 1)
                                wait_bit6[act_no] = 1;
                        if (mask[7] == 1)
                                wait_bit7[act_no] = 1;
                end
        end
end
```

```
//
////////////////////Load Mask (Mask pattern register)//////////////////////////////
//
always @ (Ld_mask)
begin
	if (Ld_mask)
		mask[0:7] = D[7:0];			//store mask pattern in mask register.
end
//
//////////Select candidates to enter next activity search logic.
//////////[0] to [7] is stimmed channel 0 to 7.
//////////[8] to [15] is waiting channel 0 to 7.
//////////[16] is suspended latch.
//////////[17] is started latch.
//////////[19] is candidate latch.
//
always @ (Suspend)							//?????
begin
	if(Suspend)
	begin
		for (i = 0; i <= 15; i = i + 1)
		begin
			if (((stim_bit0[i] == 1 & wait_bit0[i] == 1) | (stim_bit1[i] == 1 &
wait_bit1[i] == 1) | (stim_bit2[i] == 1 & wait_bit2[i] == 1) | (stim_bit3[i] == 1 & wait_bit3[i] == 1) |
(stim_bit4[i] == 1 & wait_bit4[i] == 1) | (stim_bit5[i] == 1 & wait_bit5[i] == 1) | (stim_bit6[i] == 1 &
wait_bit6[i] == 1) | (stim_bit7[i] == 1 & wait_bit7[i] == 1)) & (start_bit[i] == 1))
				begin
					cand_bit[i] = 1;
				end
				if ((suspend_bit[i] == 1) & (start_bit[i] == 1))
				begin
					cand_bit[i] = 1;
				end
		end
////////////////////////////////////////////////////////////////////////////////
//////////////////////////////Search logic//////////////////////////////////////
//
							//?????
		candfnd = 0;
		for (m = 0; m <= 15; m = m + 1)
		begin
			if (candfnd == 0)
			begin
				if (cand_bit[m] == 1)
				begin
					act_out = m;
					candfnd = 1;
				end
			end
		end
		if (candfnd == 0)
			act_out = 16;
	end
end
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
//
//////////////////////////////Overall Reset.////////////////////////////////////
//
always @ (Clrall)
```

```
begin
	if (Clrall)
	begin
		stim_bit0[0:15] = 16'b0;	//reset stim
		stim_bit1[0:15] = 16'b0;
		stim_bit2[0:15] = 16'b0;
		stim_bit3[0:15] = 16'b0;
		stim_bit4[0:15] = 16'b0;
		stim_bit5[0:15] = 16'b0;
		stim_bit6[0:15] = 16'b0;
		stim_bit7[0:15] = 16'b0;
		wait_bit0[0:15] = 16'b0;	//reset wait
		wait_bit1[0:15] = 16'b0;
		wait_bit2[0:15] = 16'b0;
		wait_bit3[0:15] = 16'b0;
		wait_bit4[0:15] = 16'b0;
		wait_bit5[0:15] = 16'b0;
		wait_bit6[0:15] = 16'b0;
		wait_bit7[0:15] = 16'b0;
		suspend_bit[0:15] = 16'b0;//reset suspend
		start_bit[0:15] = 16'b0;	//reset start
		pollend_bit[0:15] = 16'b0; //reset pollend
		cand_bit[0:15] = 16'b0;		//reset candidate
	end
end
//
//////////////////////Do external Stim (to other BUTLER).////////////////////////////
//
always @ (Do_stimx)
begin
	if (Do_stimx)
		act_x = act_no;
end
assign stimout = Do_stimx;
assign Xout = act_x;
//
//////////////////Stim from external (from other BUTLER).////////////////////////////
//
always @ (stimin)
begin
	if (~slice & stimin)
		stim_bit6[Xin] = 1;			//act no specified by Xin[3:0]
end							//channel 6 for external stim
//
////////////////////////////////Context switch./////////////////////////////////////
//
always @ (Suspend)
begin
	if (Suspend)
		slice = 1;
end
//
always @ (Do_wait)
begin
	if (Do_wait)
		slice = 1;
end
//
always @ (Nextact)
begin
```

```
        if (Nextact)
                slice = 0;
end
//
assign slice_out = slice;
//
////////////////////Ensure exactly 1 Last in every pollset////////////////////////////////
//
always @ (Suspend)                                              //?????
begin
        if(Suspend)
        begin
                Lastfnd = 0;
                for (j = 0; j <= 15; j = j + 1)                 //remove extra Last bit.
                begin
                        if (Lastfnd == 1)
                                last_bit[j] = 0;        //reset extra Last bit
                        else if (Lastfnd == 0)
                        begin
                                if (last_bit[j] == 1)           //check Last bit.
                                        Lastfnd = 1;
                        end
                        if (pollend_bit[j] == 1)        //check pollend bit.
                                Lastfnd = 0;
                end
                if (last_bit[15] == 1)
                        Lastfndup = 1;
                for (k = 14; k >= 1; k = k - 1)                 //add Last bit.
                begin
                        if (pollend_bit[k] == 1)        //check Pollend bit.
                        begin
                                if (Lastfndup == 0)
                                begin                           //set first activity in a
                                        last_bit[k+1] = 1; //pollset as Last if not found.
                                end
                                Lastfndup = 0;                  //reset Lastfndup when cross
                        end                                     //pollset boundary.
                        if (last_bit[k] == 1)                   //check Last bit.
                                Lastfndup = 1;
                end
                if (Lastfndup == 0 | pollend_bit[0] == 1)
                        last_bit[0] = 1;
        end
end
//
///////search for higher priority activity than currently running activity ///////////////
//
always @ (act_out)
begin
        if (~slice)
        begin
                if (act_out < act_run)
                        maybe_reg = 1;
        end
end
//
always @ (slice)                        //reset maybe when context switch
begin
        if (slice)
                maybe_reg = 0;
```

```
end
assign maybe = maybe_reg;
//
//////////////////////return activity to processor////////////////////////////////////
//
always @ (Nextact)
begin
	if (Nextact)
	begin
		act_run = act_out;
		if (act_out != 16)
		begin
			stim_bit0[act_run] = 0;			//reset control variables
			stim_bit1[act_run] = 0;
			stim_bit2[act_run] = 0;
			stim_bit3[act_run] = 0;
			stim_bit4[act_run] = 0;
			stim_bit5[act_run] = 0;
			stim_bit6[act_run] = 0;
			stim_bit7[act_run] = 0;
			wait_bit0[act_run] = 0;
			wait_bit1[act_run] = 0;
			wait_bit2[act_run] = 0;
			wait_bit3[act_run] = 0;
			wait_bit4[act_run] = 0;
			wait_bit5[act_run] = 0;
			wait_bit6[act_run] = 0;
			wait_bit7[act_run] = 0;
			suspend_bit[act_run] = 0;
			cand_bit[act_run] = 0;
//////////////////////////////set Last/////////////////////////////////////////////
			last_bit[act_run] = 1;				//set Last bit
			pollend_run = 0;
			for (n = act_run + 1; n <= 15; n = n + 1)	//running down
			begin
				if (pollend_run == 0)
					last_bit[n] = 0;		//reset Last bit
				if (pollend_bit[n] == 1)	//stop after pollend
					pollend_run = 1;
			end
			polltop_run = 0;
			for (p = act_run - 1; p <= 0; p = p - 1)	//running up
			begin
				if (pollend_bit[p] == 1)	//stop before pollend
					polltop_run = 1;
				if (polltop_run == 0)
					last_bit[p] = 0;	//reset Last bit
			end
		end
	end
end
//
assign Dout = act_run;
//
assign testout0 = stim_bit6;
assign testout1 = stim_bit7;
assign testout2 = suspend_bit;
assign testout3 = start_bit;
assign testout4 = pollend_bit;
assign testout5 = cand_bit;
```

```
assign testout6 = last_bit;
//
Endmodule
```

**----------Control memory (final version) ----------**

```
module Cntl_Mem(Xout, Dout, stimout, maybe, slice_out, testout0, testout1, testout2, testout3,
testout4, testout5, testout6, Xin, stimin, D, Sstart, Cstart, Spollend, Cpollend, Setsus, Do_stim,
Do_stimx, Nextact, Do_wait, Suspend, Clrall, Ld_mask, act_no, stimp0, stimp1, stimp2, stimp3);
output stimout, maybe, slice_out;
output [3:0] Xout;
output [4:0] Dout;
output [0:15] testout0, testout1, testout2, testout3, testout4, testout5, testout6;
input [3:0] act_no, Xin;
input [7:0] D;
//BUTLER instructions request signals from instruction memory
input Sstart, Cstart, Spollend, Cpollend, Setsus, Do_stim, Do_stimx, Nextact, Do_wait, Suspend,
Clrall, Ld_mask, stimin;
//stimulation from local peripherals
input stimp0, stimp1, stimp2, stimp3;
//16 activities, 8 pairs of stim-wait channels [0-7 stim, 8-15 wait]
//Suspend[16], Start[17], Pollend[18], Candidate[19], Last[20]
reg [0:15] stim_bit0, stim_bit1, stim_bit2, stim_bit3, stim_bit4, stim_bit5, stim_bit6, stim_bit7,
wait_bit0, wait_bit1, wait_bit2, wait_bit3, wait_bit4, wait_bit5, wait_bit6, wait_bit7;
reg [0:7] mask;
reg [3:0] act_x;
integer i, j, k, m, n, p, act_run, act_out, a, b, c, d, e, f, g;
reg slice, Lastfnd, Lastfndup, found, maybe_reg, pollend_run, polltop_run, searchtop, searchend,
search_start;
reg [0:15] suspend_bit, start_bit, pollend_bit, cand_bit, last_bit;
//
initial begin
act_x = 4'bzzzz;
slice = 0;
Lastfnd = 0;
Lastfndup = 0;
found = 0;
maybe_reg = 0;
pollend_run = 0;
polltop_run = 0;
searchend = 0;
searchtop = 0;
search_start = 0;
last_bit = 16'b0000000000000000;
end
//
/////////Set Start, Clear Start, Set Pollend, Clear Pollend, Set Suspend, Do Stim./////////
/////////Instruction act on activity specified in last Load Activity instruction./////////
//
always @ (Sstart)
begin
        if (Sstart)                                     //set start
                start_bit[act_no] = 1;
end
//
always @ (Cstart)
begin
        if (Cstart)                                     //clear start
                start_bit[act_no] = 0;
end
//
```

```
always @ (Spollend)
begin
	if (Spollend)				//set pollend
		pollend_bit[act_no] = 1;
end
//
always @ (Cpollend)
begin
	if (Cpollend)
		pollend_bit[act_no] = 0;
end
//
always @ (Setsus)
begin
	if (Setsus)				//set suspend
		suspend_bit[act_no] = 1;
end
//
always @ (Do_stim)
begin
	if (Do_stim)				//set stim according to mask
	begin
		if (mask[0] == 1)
			stim_bit0[act_no] = 1;
		if (mask[1] == 1)
			stim_bit1[act_no] = 1;
		if (mask[2] == 1)
			stim_bit2[act_no] = 1;
		if (mask[3] == 1)
			stim_bit3[act_no] = 1;
		if (mask[4] == 1)
			stim_bit4[act_no] = 1;
		if (mask[5] == 1)
			stim_bit5[act_no] = 1;
		if (mask[6] == 1)
			stim_bit6[act_no] = 1;
		if (mask[7] == 1)
			stim_bit7[act_no] = 1;
	end
end
//
////////////////////////////Set stim by local peripherals////////////////////////////
always @ (stimp0)
begin
	if (stimp0)
	begin
		wait (~slice)
		begin
			stim_bit7[0] = 1;
			stim_bit7[4] = 1;
			stim_bit7[8] = 1;
			stim_bit7[12] = 1;
		end
	end
end
//
always @ (stimp1)
begin
	if (stimp1)
	begin
```

```
                wait (~slice)
                begin
                        stim_bit7[1] = 1;
                        stim_bit7[5] = 1;
                        stim_bit7[9] = 1;
                        stim_bit7[13] = 1;
                end
        end
end
//
always @ (stimp2)
begin
        if (stimp2)
        begin
                wait (~slice)
                begin
                        stim_bit7[2] = 1;
                        stim_bit7[6] = 1;
                        stim_bit7[10] = 1;
                        stim_bit7[14] = 1;
                end
        end
end
//
always @ (stimp3)
begin
        if (stimp3)
        begin
                wait (~slice)
                begin
                        stim_bit7[3] = 1;
                        stim_bit7[7] = 1;
                        stim_bit7[11] = 1;
                        stim_bit7[15] = 1;
                end
        end
end
//
//
////////////////////////////Suspend, Do Wait.////////////////////////////////////
//////////instruction act on activity currently running in the processor.//////////
//
always @ (Suspend)
begin
        if (Suspend)
        begin
                if (act_run != 16)
                        suspend_bit[act_run] = 1;
        end
end
//
always @ (Do_wait)
begin
        if (Do_wait)                                            //set wait according to mask.
        begin
                if (act_run != 16)
                begin
                        if (mask[0] == 1)
                                wait_bit0[act_no] = 1;
                        if (mask[1] == 1)
```

```
                                wait_bit1[act_no] = 1;
                        if (mask[2] == 1)
                                wait_bit2[act_no] = 1;
                        if (mask[3] == 1)
                                wait_bit3[act_no] = 1;
                        if (mask[4] == 1)
                                wait_bit4[act_no] = 1;
                        if (mask[5] == 1)
                                wait_bit5[act_no] = 1;
                        if (mask[6] == 1)
                                wait_bit6[act_no] = 1;
                        if (mask[7] == 1)
                                wait_bit7[act_no] = 1;
                end
        end
end
//
////////////////////Load Mask (Mask pattern register)//////////////////////////////////
//
always @ (Ld_mask)
begin
        if (Ld_mask)
                mask[0:7] = D[7:0];                     //store mask pattern in mask register.
end
//
//////////Select candidates to enter next activity search logic.
//////////[0] to [7] is stimmed channel 0 to 7.
//////////[8] to [15] is waiting channel 0 to 7.
//////////[16] is suspended latch.
//////////[17] is started latch.
//////////[19] is candidate latch.
//
always @ (Suspend)                                              //?????
begin
        if(~Suspend)
        begin
                for (i = 0; i <= 15; i = i + 1)
                begin
                        if (((stim_bit0[i] == 1 & wait_bit0[i] == 1) | (stim_bit1[i] == 1 &
wait_bit1[i] == 1) | (stim_bit2[i] == 1 & wait_bit2[i] == 1) | (stim_bit3[i] == 1 & wait_bit3[i] == 1) |
(stim_bit4[i] == 1 & wait_bit4[i] == 1) | (stim_bit5[i] == 1 & wait_bit5[i] == 1) | (stim_bit6[i] == 1 &
wait_bit6[i] == 1) | (stim_bit7[i] == 1 & wait_bit7[i] == 1)) & (start_bit[i] == 1))
                        begin
                                cand_bit[i] = 1;
                        end
                        if ((suspend_bit[i] == 1) & (start_bit[i] == 1))
                        begin
                                cand_bit[i] = 1;
                        end
                end
////////////////////////////////////////////////////////////////////////////
//////////////////////////////////Search logic//////////////////////////////////
//
////////////////////////////////search act[0]//////////////////////////////////
                found = 0;                                      //reset in each pollset
                if (last_bit[0] == 1)                           //last?
                begin
//////////////////////////////Round Robin/////////////////////////////////////
                        if (pollend_bit[0] == 0)                //pollend?
                        begin
```

```
                                searchend = 0;
                                for (a = 1; a <= 15; a = a + 1)
                                begin
                                        if ((searchend == 0) & (found == 0))
                                        begin
                                                if (cand_bit[a] == 1)           //candidate?
                                                begin
                                                        act_out = a;
                                                        found = 1;
                                                end
                                                if (pollend_bit[a] == 1)        //pollend?
                                                        searchend = 1;
                                        end
                                end
                                if (found == 0)
                                begin
                                        if (cand_bit[0] == 1)
                                        begin
                                                act_out = 0;
                                                found = 1;
                                        end
                                end
                        end
                        else if (pollend_bit[0] == 1)                           //pollend?
                        begin
                                if (cand_bit[0] == 1)                           //candidate?
                                begin
                                        act_out = 0;
                                        found = 1;
                                end
                        end
                end
////////////////////////////search act[1] to act[14]/////////////////////////////////////
                for (b = 1; b <= 14; b = b + 1)
                begin
                        if(found == 0)
                        begin
                                if(last_bit[b] == 1)
        //last?
                                begin
//Round Robin
                                        if (pollend_bit[b] == 0)
        //pollend?
                                        begin
                                                searchend = 0;
                                                for (c = b + 1; c <= 15; c = c + 1)
                                                begin
                                                        if (searchend == 0 & found == 0)
                                                        begin
                                                                if (cand_bit[c] == 1)
        //candidate?
                                                                begin
                                                                        act_out = c;
                                                                        found = 1;
                                                                end
                                                                if (pollend_bit[c] == 1)
        //pollend?
                                                                        searchend = 1;
                                                        end
                                                end
```

```
                                        if (found == 0)
                                        begin
                                                searchtop = 0;
                                                for (d = b; d >= 0; d = d - 1)
                                                begin
                                                        if (pollend_bit[d] == 1)
        //pollend?
                                                                searchtop = 1;
                                                        if (searchtop == 0)
                                                        begin
                                                                if (cand_bit[d] == 1)
        //candidate?
                                                                begin
                                                                        act_out = d;
                                                                        found = 1;
                                                                end
                                                        end
                                                end
                                        end
                                        else if (pollend_bit[b] == 1)
                                        begin
                                                if (found == 0)
                                                begin
                                                        searchtop = 0;
                                                        for (e = b - 1; e >= 0; e = e - 1)
                                                        begin
                                                                if (pollend_bit[e] == 1)
        //pollend?
                                                                searchtop = 1;
                                                        if (searchtop == 0)
                                                        begin
                                                                if (cand_bit[e] == 1)
                //candidate?
                                                                begin
                                                                        act_out = e;
                                                                        found = 1;
                                                                end
                                                        end
                                                end
                                        end
                                        if (found == 0)
                                        begin
                                                if (cand_bit[b] == 1)
                                                begin
                                                        act_out = b;
                                                        found = 1;
                                                end
                                        end
                                end

                        end
                end
        end
////////////////////////////////////search act[15]////////////////////////////////////
        if (found == 0)
        begin
                if (last_bit[15] == 1)
        //last?
                begin
```

```
                                if (pollend_bit[15] == 1)                       //pollend?
                                begin
                                        searchtop = 0;
                                        for (f = 14; f >= 0; f = f - 1)
                                        begin
                                                if (pollend_bit[f] == 1)        //pollend?
                                                        searchtop = 1;
                                                if (searchtop == 0)
                                                begin
                                                        if (cand_bit[f] == 1)
        //candidate?
                                                        begin
                                                                act_out = f;
                                                                found = 1;
                                                        end
                                                end
                                        end
                                end
                                if (found == 0)
                                begin
                                        if (cand_bit[15] == 1)
                                        begin
                                                act_out = 15;
                                                found = 1;
                                        end
                                end
                                else if (pollend_bit[15] == 0)
        //pollend?
                                begin
                                        searchtop = 0;
                                        for(g = 15; g >= 0; g = g - 1)
                                        begin
                                                if (pollend_bit[g] == 1)        //pollend?
                                                        searchtop = 1;
                                                if (searchtop == 0)
                                                begin
                                                        if (cand_bit[g] == 1)
        //candidate?
                                                        begin
                                                                act_out = g;
                                                                found = 1;
                                                        end
                                                end
                                        end
                                end
                        end
                end
////////////////////////return act[16] if no candidate found////////////////////////////////
                if (found == 0)
                        act_out = 16;
        end
end
//
//////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
//
////////////////////////////////Overall Reset.////////////////////////////////////
//
always @ (Clrall)
begin
```

```
	if (Clrall)
	begin
		stim_bit0[0:15] = 16'b0;	//reset stim
		stim_bit1[0:15] = 16'b0;
		stim_bit2[0:15] = 16'b0;
		stim_bit3[0:15] = 16'b0;
		stim_bit4[0:15] = 16'b0;
		stim_bit5[0:15] = 16'b0;
		stim_bit6[0:15] = 16'b0;
		stim_bit7[0:15] = 16'b0;
		wait_bit0[0:15] = 16'b0;	//reset wait
		wait_bit1[0:15] = 16'b0;
		wait_bit2[0:15] = 16'b0;
		wait_bit3[0:15] = 16'b0;
		wait_bit4[0:15] = 16'b0;
		wait_bit5[0:15] = 16'b0;
		wait_bit6[0:15] = 16'b0;
		wait_bit7[0:15] = 16'b0;
		suspend_bit[0:15] = 16'b0;//reset suspend
		start_bit[0:15] = 16'b0;	//reset start
		pollend_bit[0:15] = 16'b0; //reset pollend
		cand_bit[0:15] = 16'b0;		//reset candidate
	end
end
//
//////////////////////Do external Stim (to other BUTLER).////////////////////////////
//
always @ (Do_stimx)
begin
	if (Do_stimx)
		act_x = act_no;
end
assign stimout = Do_stimx;
assign Xout = act_x;
//
//////////////////Stim from external (from other BUTLER).////////////////////////////
//
always @ (stimin)
begin
	if (~slice & stimin)
		stim_bit6[Xin] = 1;			//act no specified by Xin[3:0]
end							//channel 6 for external stim
//
////////////////////////////////Context switch./////////////////////////////////////
//
always @ (Suspend)
begin
	if (Suspend)
		slice = 1;
end
//
always @ (Do_wait)
begin
	if (Do_wait)
		slice = 1;
end
//
always @ (Nextact)
begin
	if (Nextact)
```

```
                slice = 0;
end
//
assign slice_out = slice;
//
////////////////////Ensure exactly 1 Last in every pollset////////////////////////////////
//
always @ (Suspend)                                                      //?????
begin
        if(Suspend)
        begin
                Lastfnd = 0;
                for (j = 0; j <= 15; j = j + 1)                         //remove extra Last bit.
                begin
                        if (Lastfnd == 1)
                                last_bit[j] = 0;                //reset extra Last bit
                        else if (Lastfnd == 0)
                        begin
                                if (last_bit[j] == 1)                   //check Last bit.
                                        Lastfnd = 1;
                        end
                        if (pollend_bit[j] == 1)                //check pollend bit.
                                Lastfnd = 0;
                end
                Lastfndup = 0;
                if (last_bit[15] == 1)
                        Lastfndup = 1;
                for (k = 14; k >= 1; k = k - 1)                         //add Last bit.
                begin
                        if (pollend_bit[k] == 1)                //check Pollend bit.
                        begin
                                if (Lastfndup == 0)
                                begin                                   //set first activity in a
                                        last_bit[k+1] = 1; //pollset as Last if not found.
                                end
                                Lastfndup = 0;                          //reset Lastfndup when cross
                        end                                             //pollset boundary.
                        if (last_bit[k] == 1)                           //check Last bit.
                                Lastfndup = 1;
                end
                if ((Lastfndup == 0) | (pollend_bit[0] == 1))
                        last_bit[0] = 1;
        end
end
//
///////search for higher priority activity than currently running activity ////////////////////////////////not
checked!!!////////
//
always @ (act_out)
begin
        if(~slice)
        begin
                search_start = 0;
                if (act_run != 0)
                begin
                        for (m = act_run - 1; m >= 0; m = m - 1)
                        begin
                                if (pollend_bit[m] == 1)
                                        search_start = 1;
                                if (search_start == 1)
```

```
                                        begin
                                                if (cand_bit[m] == 1)
                                                        maybe_reg = 1;
                                        end
                                end
                        end
                end
end
//
always @ (slice)                                        //reset maybe when context switch
begin
        if (slice)
                maybe_reg = 0;
end
assign maybe = maybe_reg;
//
//////////////////////return activity to processor/////////////////////////////////////
//
always @ (Nextact)
begin
        if (Nextact)
        begin
                act_run = act_out;
                if (act_out != 16)
                begin
                        stim_bit0[act_run] = 0;                         //reset control variables
                        stim_bit1[act_run] = 0;
                        stim_bit2[act_run] = 0;
                        stim_bit3[act_run] = 0;
                        stim_bit4[act_run] = 0;
                        stim_bit5[act_run] = 0;
                        stim_bit6[act_run] = 0;
                        stim_bit7[act_run] = 0;
                        wait_bit0[act_run] = 0;
                        wait_bit1[act_run] = 0;
                        wait_bit2[act_run] = 0;
                        wait_bit3[act_run] = 0;
                        wait_bit4[act_run] = 0;
                        wait_bit5[act_run] = 0;
                        wait_bit6[act_run] = 0;
                        wait_bit7[act_run] = 0;
                        suspend_bit[act_run] = 0;
                        cand_bit[act_run] = 0;
//////////////////////////////////set Last/////////////////////////////////////////////
                        last_bit[act_run] = 1;                                  //set Last bit

                        if (act_run != 0)
                        begin
                                polltop_run = 0;
                                for (p = act_run - 1; p >= 0; p = p - 1)        //running up
                                begin
                                        if (pollend_bit[p] == 1)                //stop before pollend
                                                polltop_run = 1;
                                        if (polltop_run == 0)
                                                last_bit[p] = 0;                //reset Last bit
                                end
                        end
                        if (act_run != 15)
                        begin
                                if (pollend_bit[act_run] == 0)
```

```
                                        begin
                                                pollend_run = 0;
                                                for (n = act_run + 1; n <= 15; n = n + 1)       //running
down
                                                begin
                                                        if (pollend_run == 0)
                                                                last_bit[n] = 0;                //reset Last
bit
                                                        if (pollend_bit[n] == 1)                //stop after
pollend
                                                                pollend_run = 1;
                                                end
                                        end
                                end
                        end
                end
        end
end
//
assign Dout = act_run;
//
assign testout0 = stim_bit6;
assign testout1 = stim_bit7;
assign testout2 = suspend_bit;
assign testout3 = start_bit;
assign testout4 = pollend_bit;
assign testout5 = cand_bit;
assign testout6 = last_bit;
//
endmodule
```

Appendix 2

```
---------------------------- Interrupt controller simulation -------------------------------
Initial begin
Clrall = 0;
Test = 0;
act_no = 4'bzzzz;
slice = 0;
maybe = 0;
expired = 0;
#2      act_no = 4'b0011;
#1      Test = 1;
#1      Test = 0;
#2      maybe = 1;
#2      maybe = 0;
#2      expired = 1;
#2      slice = 1;
#2      expired = 0;
#2      slice = 0;
#2      maybe = 1;
#2      act_no = 4'b0001;
#1      Test = 1;
#1      Test = 0;
#2      expired = 1;
#2      act_no = 4'b0000;
#1      Test = 1;
#1      Test = 0;
End
```

```
---------------------------------- Decoder simulation ----------------------------------------
initial begin
R = 0;
W = 0;
select = 0;
a0 = 0;
a1 = 0;
a2 = 0;
//read cycle
#2 R = 1;
#1 select = 1;
#2 a2 = 1;
#2 a1 = 1;
#2 a2 = 0;
#2 a0 = 1;
#2 a1 = 0;
#2 a2 = 1;
#2 a1 = 1;
#2 a0 = 0;
  a1 = 0;
  a2 = 0;
  R = 0;
//write cycle
#1 W = 1;
#1 select = 1;
#2 a2 = 1;
#2 a1 = 1;
#2 a2 = 0;
#2 a0 = 1;
#2 a1 = 0;
#2 a2 = 1;
#2 a1 = 1;
#2 select = 0;
```

```
End
---------------------------------------- Counter simulation ----------------------------------------
initial begin
Test = 0;
slice = 0;
D[7:0] = 8'b0;
countin = 0;
Ld_cntr_lo = 0;
Ld_cntr_hi = 0;
//Load counter
#2      D[7:0] = 8'b11111011;
#2      Ld_cntr_lo = 1;
#2      Ld_cntr_lo = 0;
        D[7:0] = 8'b11111111;
#2      Ld_cntr_hi = 1;
#2      Ld_cntr_hi = 0;
//initialize counter
#5      slice = 1;
#5      slice = 0;
//start counting
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
//reset counter
#5      slice = 1;
#5      slice = 0;
End
---------------------- Activity number register simulation ----------------------------------
initial begin
Ld_act = 0;
D[3:0] = 4'b0;
//
#2      Ld_act = 1;
#2      Ld_act = 0;
#4      D[3:0] = 4'b1010;
#2      Ld_act = 1;
#2      Ld_act = 0;
#2      D[3:0] = 4'b1110;
#2      D[3:0] = 4'b0101;
#2      Ld_act = 1;
#3      Ld_act = 0;
End
------------------------- Control memory simulation (simple version) ----------------------
```

```
initial begin
D[7:0] = 8'bzzzzzzzz;
Clrall = 0;
Cpollend = 0;
Cstart = 0;
Do_stim = 0;
Do_stimx = 0;
Do_wait = 0;
Ld_mask = 0;
Nextact = 0;
Setsus = 0;
Spollend = 0;
Sstart = 0;
Suspend = 0;
Xin[3:0] = 4'bzzzz;
act_no[3:0] = 4'bzzzz;
stimin = 0;
stimp0 = 0;
stimp1 = 0;
stimp2 = 0;
stimp3 = 0;
//
#1      Clrall = 1;
#1      Clrall = 0;
#2      D[7:0] = 8'b11111111;
#1      Ld_mask = 1;                    //test Ld_mask
#1      Ld_mask = 0;
#2      act_no[3:0] = 4'b0100;
#2      Sstart = 1;                     //test Sstart
#1      Sstart = 0;
#2      Cstart = 1;                     //test Cstart
#1      Cstart = 0;
#2      Spollend = 1;                   //test Sstart
#1      Spollend = 0;
#2      Cpollend = 1;                   //test Cstart
#1      Cpollend = 0;
#2      Setsus = 1;                     //test Setsus
#1      Setsus = 0;
#2      Do_stim = 1;                    //test Do_stim
#2      Do_stim = 0;
#2      stimp0 = 1;                     //stimp between context switch
#2      stimp0 = 0;
#1      Sstart = 1;
#1      Sstart = 0;
#1      Suspend = 1;
#2      Suspend = 0;
#2      stimp1 = 1;                     //stimp during context switch
#2      stimp1 = 0;
#2      Nextact = 1;                    //return act[0100]
#1      Nextact = 0;
#2      Suspend = 1;                    //test Suspend
#2      Suspend = 0;
#1      Nextact = 1;
#2      Nextact = 0;
#2      Do_wait = 1;                    //test Do_wait
#2      Do_wait = 0;
#1      Nextact = 1;
#2      Nextact = 0;
#2      Clrall = 1;                     //test Clrall
#2      Clrall = 0;
```

```
#1      act_no[3:0] = 4'b1010;
#1      Do_stimx = 1;                   //test Do_stimx
#1      Do_stimx = 0;
#2      Xin[3:0] = 4'b0100;             //test stimin act[Xin]
#1      stimin = 1;
#2      stimin = 0;
End
```

---

```
---------------------- Control memory simulation (final version) ----------------------------
initial begin
D[7:0] = 8'bzzzzzzzz;
Clrall = 0;
Cpollend = 0;
Cstart = 0;
Do_stim = 0;
Do_stimx = 0;
Do_wait = 0;
Ld_mask = 0;
Nextact = 0;
Setsus = 0;
Spollend = 0;
Sstart = 0;
Suspend = 0;
Xin[3:0] = 4'bzzzz;
act_no[3:0] = 4'bzzzz;
stimin = 0;
stimp0 = 0;
stimp1 = 0;
stimp2 = 0;
stimp3 = 0;
//
////////////////////////////////////////
#1      Clrall = 1;                     //Clrall
#1      Clrall = 0;
//////////////////////////////////////////test Round Robin
#1      act_no[3:0] = 4'b1110;
#2      act_no[3:0] = 4'b0100;
#2      Sstart = 1;                     //Sstart
#1      Sstart = 0;
#2      Setsus = 1;                     //Setsus
#1      Setsus = 0;
#2      act_no[3:0] = 4'b0101;
#2      Sstart = 1;                     //Sstart
#1      Sstart = 0;
#2      Setsus = 1;                     //Setsus
#1      Setsus = 0;
#2      act_no[3:0] = 4'b0110;
#2      Sstart = 1;                     //Sstart
#1      Sstart = 0;
#2      Setsus = 1;                     //Setsus
#1      Setsus = 0;
#2      Suspend = 1;
#1      Suspend = 0;
#5      Nextact = 1;                    //Nextact
#1      Nextact = 0;
#1      act_no[3:0] = 4'b0010;
#2      Spollend = 1;                   //Spollend
#2      Spollend = 0;
#1      act_no[3:0] = 4'b1110;
#2      Spollend = 1;                   //Spollend
#2      Spollend = 0;
```

```
#2      Suspend = 1;
#1      Suspend = 0;
#5      Nextact = 1;                    //Nextact
#1      Nextact = 0;
#2      Suspend = 1;
#1      Suspend = 0;
#5      Nextact = 1;                    //Nextact
#1      Nextact = 0;
#2      Suspend = 1;
#1      Suspend = 0;
#5      Nextact = 1;                    //Nextact
#1      Nextact = 0;
//////////////////////////////////////////test maybe search
#2      act_no[3:0] = 4'b0000;
#2      Sstart = 1;                     //Sstart
#1      Sstart = 0;
#2      Setsus = 1;                     //Setsus
#1      Setsus = 0;
//////////////////////////////////////////return act[0000]
#2      Suspend = 1;
#2      Suspend = 0;
#5      Nextact = 1;
#2      Nextact = 0;
#2      Do_wait = 1;
#2      Do_wait = 0;
//////////////////////////////////////////test 1 "last" logic
#1      act_no[3:0] = 4'b0010;
#2      Cpollend = 1;                   //Cpollend
#2      Cpollend = 0;
//////////////////////////////////////////test return idle
#2      Clrall = 1;
#2      Clrall = 0;
#2      Suspend = 1;
#2      Suspend = 0;
#2      Nextact = 1;
#2      Nextact = 0;
End
```

---------- Integrated simulation ----------

```
initial begin
D[7:0] = 8'bzzzzzzzz;
a0 = 0;
a1 = 0;
a2 = 0;
R = 0;
W = 0;
select = 0;
Xin[3:0] = 4'bzzzz;
stimin = 0;
stimp0 = 0;
stimp1 = 0;
stimp2 = 0;
stimp3 = 0;
countin = 1'bz;
////////////////////////////////////////////////////////////////Interrupt controller
//
#2      D[3:0] = 4'b0011;           //Ld_act
#1      a0 = 0;
        a1 = 0;
        a2 = 1;
        W = 1;
```

```
#1      select = 1;
#1      select = 0;
        W = 0;
//
#1      a0 = 1;                         //Test
        a1 = 1;
        a2 = 1;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
////////////////////////////////////////
//Load counter
#2      D[7:0] = 8'b11111011;
//
#1      a0 = 1;                         //Ld_cntr_lo
        a1 = 1;
        a2 = 0;
        W = 1;
#1      select = 1;
#1      select = 0;
        W = 0;
//
        D[7:0] = 8'b11111111;
//
#1      a0 = 1;                         //Ld_cntr_hi
        a1 = 1;
        a2 = 1;
        W = 1;
#1      select = 1;
#1      select = 0;
        W = 0;
//
//initialize counter
//
#1      a0 = 0;                         //Suspend
        a1 = 1;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#1      a0 = 1;                         //Nextact
        a1 = 1;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
////////////////////////////////////////expired
//start counting
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
```

```
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
//////////////////////////////////////////////reset by slice
//
#1      a0 = 0;                         //Suspend
        a1 = 1;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
//
#1      a0 = 1;                         //Nextact
        a1 = 1;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//////////////////////////////////////////////expired
//start counting
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
//////////////////////////////////////////////reset by actbit = 0
#2      D[3:0] = 4'b0000;        //Ld_act
#1      a0 = 0;
        a1 = 0;
        a2 = 1;
        W = 1;
#1      select = 1;
```

```
#1      select = 0;
        W = 0;
//
#1      a0 = 1;                         //Test
        a1 = 1;
        a2 = 1;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
////////////////////////////////////////expired
//initialize counter
//
#1      a0 = 0;                         //Suspend
        a1 = 1;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#1      a0 = 1;                         //Nextact
        a1 = 1;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
//start counting
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
////////////////////////////////////////////
//
#2      D[3:0] = 4'b0000;           //Ld_act
#1      a0 = 0;
        a1 = 0;
        a2 = 1;
        W = 1;
#1      select = 1;
#1      select = 0;
```

```
        W = 0;
//
#1      a0 = 1;                         //Test
        a1 = 1;
        a2 = 1;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
////////////////////////////////////////reset counter
//
#1      a0 = 0;                         //Suspend
        a1 = 1;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#1      a0 = 1;                         //Nextact
        a1 = 1;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
////////////////////////////////////////////////
//
#1      a0 = 0;                         //Clrall
        a1 = 1;
        a2 = 1;
        W = 1;
#1      select = 1;
#1      select = 0;
        W = 0;
//
////////////////////////////////////////////////////////////////////////////Counter
//Load counter
#2      D[7:0] = 8'b11111011;
//
#1      a0 = 1;                         //Ld_cntr_lo
        a1 = 1;
        a2 = 0;
        W = 1;
#1      select = 1;
#1      select = 0;
        W = 0;
//
        D[7:0] = 8'b11111111;
//
#1      a0 = 1;                         //Ld_cntr_hi
        a1 = 1;
        a2 = 1;
        W = 1;
#1      select = 1;
#1      select = 0;
        W = 0;
//
//initialize counter
```

```
//
#1      a0 = 0;                         //Suspend
        a1 = 1;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#1      a0 = 1;                         //Nextact
        a1 = 1;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
//start counting
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
#1      countin = 1;
#1      countin = 0;
//reset counter
//
#1      a0 = 0;                         //Suspend
        a1 = 1;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#1      a0 = 1;                         //Nextact
        a1 = 1;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
////////////////////////////////////////////////////////////////////////////simple version cntl mem
//
#1      a0 = 0;                         //Clrall
        a1 = 1;
```

```
        a2 = 1;
        W = 1;
#1      select = 1;
#1      select = 0;
        W = 0;
//
#2      D[7:0] = 8'b11111111;
//
#1      a0 = 0;                         //Ld_mask
        a1 = 0;
        a2 = 0;
        W = 1;
#1      select = 1;
#1      select = 0;
        W = 0;
//
#2      D[3:0] = 4'b0100;        //Ld_act
#1      a0 = 0;
        a1 = 0;
        a2 = 1;
        W = 1;
#1      select = 1;
#1      select = 0;
        W = 0;
//
#1      a0 = 1;                         //Sstart
        a1 = 0;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#1      a0 = 1;                         //Cstart
        a1 = 0;
        a2 = 0;
        W = 1;
#1      select = 1;
#1      select = 0;
        W = 0;
//
#1      a0 = 1;                         //Spollend
        a1 = 0;
        a2 = 1;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#1      a0 = 1;                         //Cpollend
        a1 = 0;
        a2 = 1;
        W = 1;
#1      select = 1;
#1      select = 0;
        W = 0;
//
#1      a0 = 0;                         //Setsus
        a1 = 1;
        a2 = 1;
```

```
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#1      a0 = 0;                         //Do_stim
        a1 = 0;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#2      stimp0 = 1;                     //stimp between context switch
#2      stimp0 = 0;
//
#1      a0 = 1;                         //Sstart
        a1 = 0;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#1      a0 = 0;                         //Suspend
        a1 = 1;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#2      stimp1 = 1;                     //stimp during context switch
#2      stimp1 = 0;
//
#1      a0 = 1;                         //Nextact
        a1 = 1;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#1      a0 = 0;                         //Suspend
        a1 = 1;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#1      a0 = 1;                         //Nextact
        a1 = 1;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#1      a0 = 0;                         //Do_wait
```

```
        a1 = 0;
        a2 = 1;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#1      a0 = 1;                         //Nextact
        a1 = 1;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#1      a0 = 0;                         //Clrall
        a1 = 1;
        a2 = 1;
        W = 1;
#1      select = 1;
#1      select = 0;
        W = 0;
//
#2      D[3:0] = 4'b1010;       //Ld_act
#1      a0 = 0;
        a1 = 0;
        a2 = 1;
        W = 1;
#1      select = 1;
#1      select = 0;
        W = 0;
//
#1      a0 = 0;                         //Do_stimx
        a1 = 1;
        a2 = 0;
        W = 1;
#1      select = 1;
#1      select = 0;
        W = 0;
//
#2      Xin[3:0] = 4'b0100;             //test stimin act[Xin]
#1      stimin = 1;
#2      stimin = 0;
////////////////////////////////////////////////////////////////////////////Final version cntl mem
#1      a0 = 0;                         //Clrall
        a1 = 1;
        a2 = 1;
        W = 1;
#1      select = 1;
#1      select = 0;
        W = 0;
//////////////////////////////////////////test Round Robin
//
#2      D[3:0] = 4'b0100;       //Ld_act
#1      a0 = 0;
        a1 = 0;
        a2 = 1;
        W = 1;
#1      select = 1;
#1      select = 0;
```

```
        W = 0;
//
#1      a0 = 1;                         //Sstart
        a1 = 0;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#1      a0 = 0;                         //Setsus
        a1 = 1;
        a2 = 1;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#2      D[3:0] = 4'b0101;        //Ld_act
#1      a0 = 0;
        a1 = 0;
        a2 = 1;
        W = 1;
#1      select = 1;
#1      select = 0;
        W = 0;
//
#1      a0 = 1;                         //Sstart
        a1 = 0;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#1      a0 = 0;                         //Setsus
        a1 = 1;
        a2 = 1;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#2      D[3:0] = 4'b0110;        //Ld_act
#1      a0 = 0;
        a1 = 0;
        a2 = 1;
        W = 1;
#1      select = 1;
#1      select = 0;
        W = 0;
//
#1      a0 = 1;                         //Sstart
        a1 = 0;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
```

```
#1      a0 = 0;                         //Setsus
        a1 = 1;
        a2 = 1;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#1      a0 = 0;                         //Suspend
        a1 = 1;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#1      a0 = 1;                         //Nextact
        a1 = 1;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#2      D[3:0] = 4'b0010;        //Ld_act
#1      a0 = 0;
        a1 = 0;
        a2 = 1;
        W = 1;
#1      select = 1;
#1      select = 0;
        W = 0;
//
#1      a0 = 1;                         //Spollend
        a1 = 0;
        a2 = 1;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#2      D[3:0] = 4'b1110;        //Ld_act
#1      a0 = 0;
        a1 = 0;
        a2 = 1;
        W = 1;
#1      select = 1;
#1      select = 0;
        W = 0;
//
#1      a0 = 1;                         //Spollend
        a1 = 0;
        a2 = 1;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#1      a0 = 0;                         //Suspend
        a1 = 1;
```

```
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#1      a0 = 1;                         //Nextact
        a1 = 1;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#1      a0 = 0;                         //Suspend
        a1 = 1;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#1      a0 = 1;                         //Nextact
        a1 = 1;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#1      a0 = 0;                         //Suspend
        a1 = 1;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#1      a0 = 1;                         //Nextact
        a1 = 1;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
////////////////////////////////////////test maybe search
//
#2      D[3:0] = 4'b0000;           //Ld_act
#1      a0 = 0;
        a1 = 0;
        a2 = 1;
        W = 1;
#1      select = 1;
#1      select = 0;
        W = 0;
//
#1      a0 = 1;                         //Sstart
        a1 = 0;
        a2 = 0;
```

```
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#1      a0 = 0;                         //Setsus
        a1 = 1;
        a2 = 1;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
//////////////////////////////////////////return act[0000]
#1      a0 = 0;                         //Suspend
        a1 = 1;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#1      a0 = 1;                         //Nextact
        a1 = 1;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#1      a0 = 0;                         //Do_wait
        a1 = 0;
        a2 = 1;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
//////////////////////////////////////////test 1 "last" logic
//
#2      D[3:0] = 4'b0010;           //Ld_act
#1      a0 = 0;
        a1 = 0;
        a2 = 1;
        W = 1;
#1      select = 1;
#1      select = 0;
        W = 0;
//
#1      a0 = 1;                         //Cpollend
        a1 = 0;
        a2 = 1;
        W = 1;
#1      select = 1;
#1      select = 0;
        W = 0;
//
//////////////////////////////////////////test return idle
#1      a0 = 0;                         //Clrall
        a1 = 1;
```

```
        a2 = 1;
        W = 1;
#1      select = 1;
#1      select = 0;
        W = 0;
//
#1      a0 = 0;                         //Suspend
        a1 = 1;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
#1      a0 = 1;                         //Nextact
        a1 = 1;
        a2 = 0;
        R = 1;
#1      select = 1;
#1      select = 0;
        R = 0;
//
End
```

**Index of Figures and Tables**

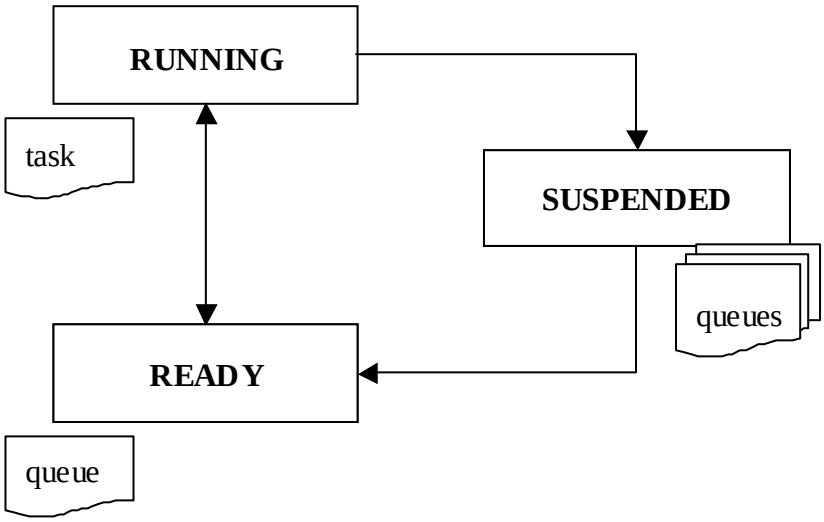



Fig. 1 Task States

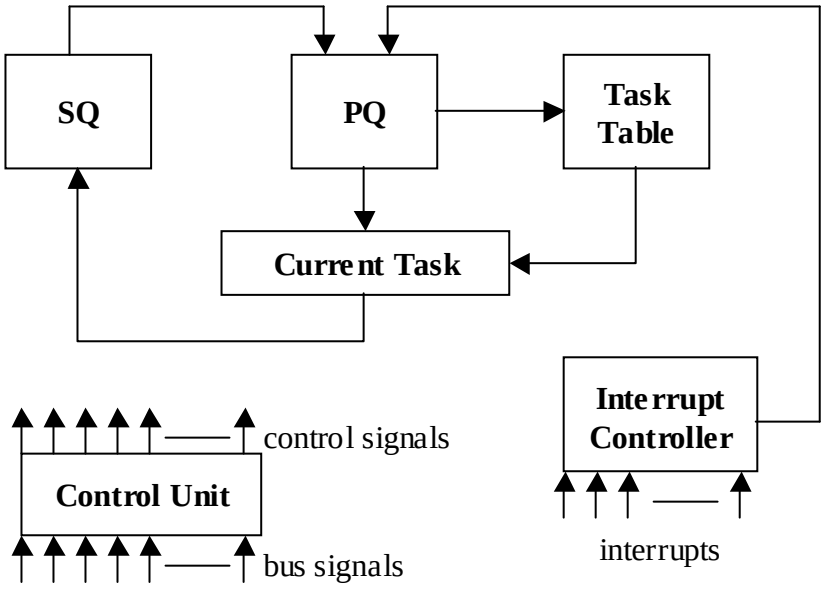



Fig. 2 The configurable hardware scheduler micro-architecture

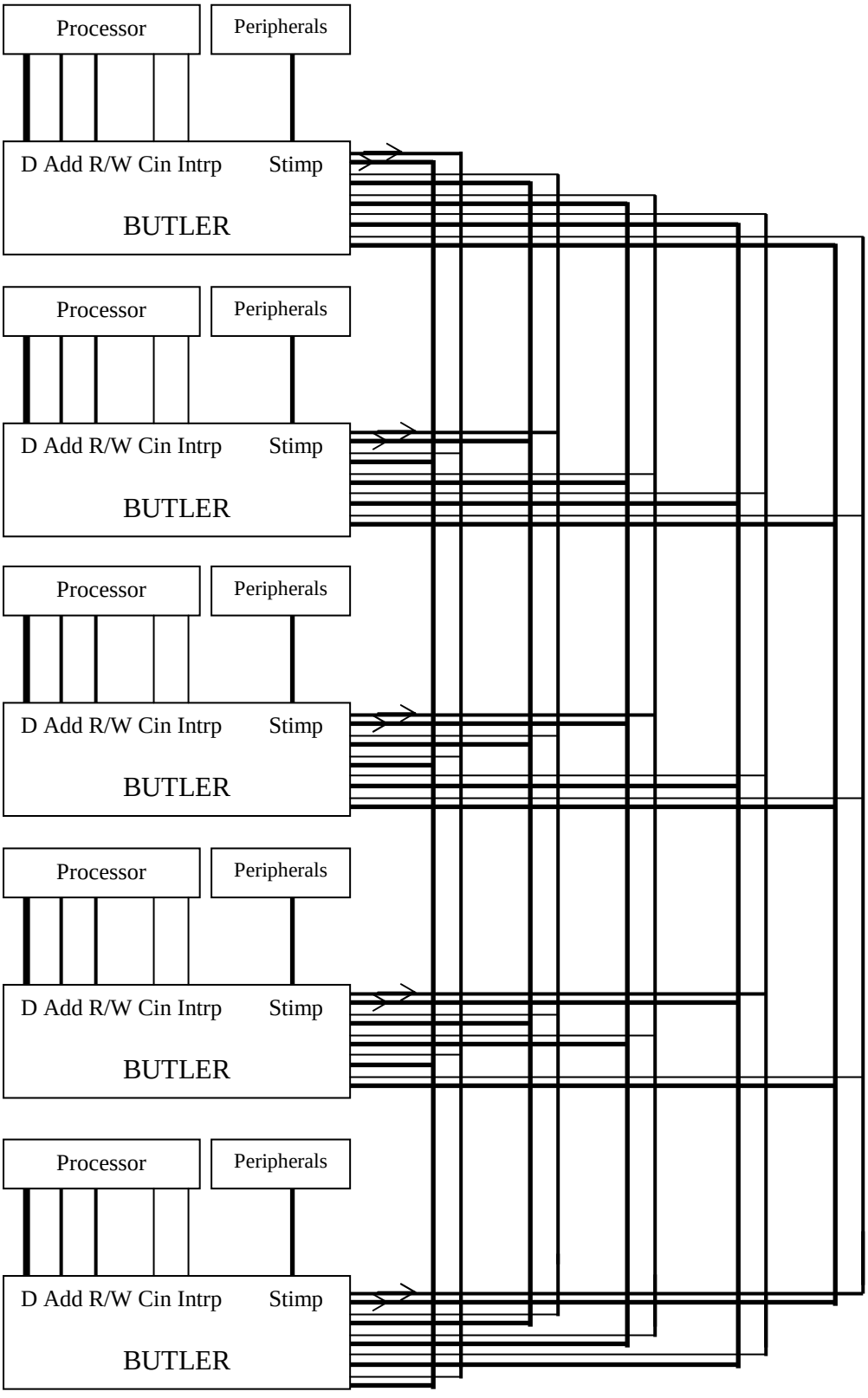



Fig. 3 BUTLER System Overall Configuration

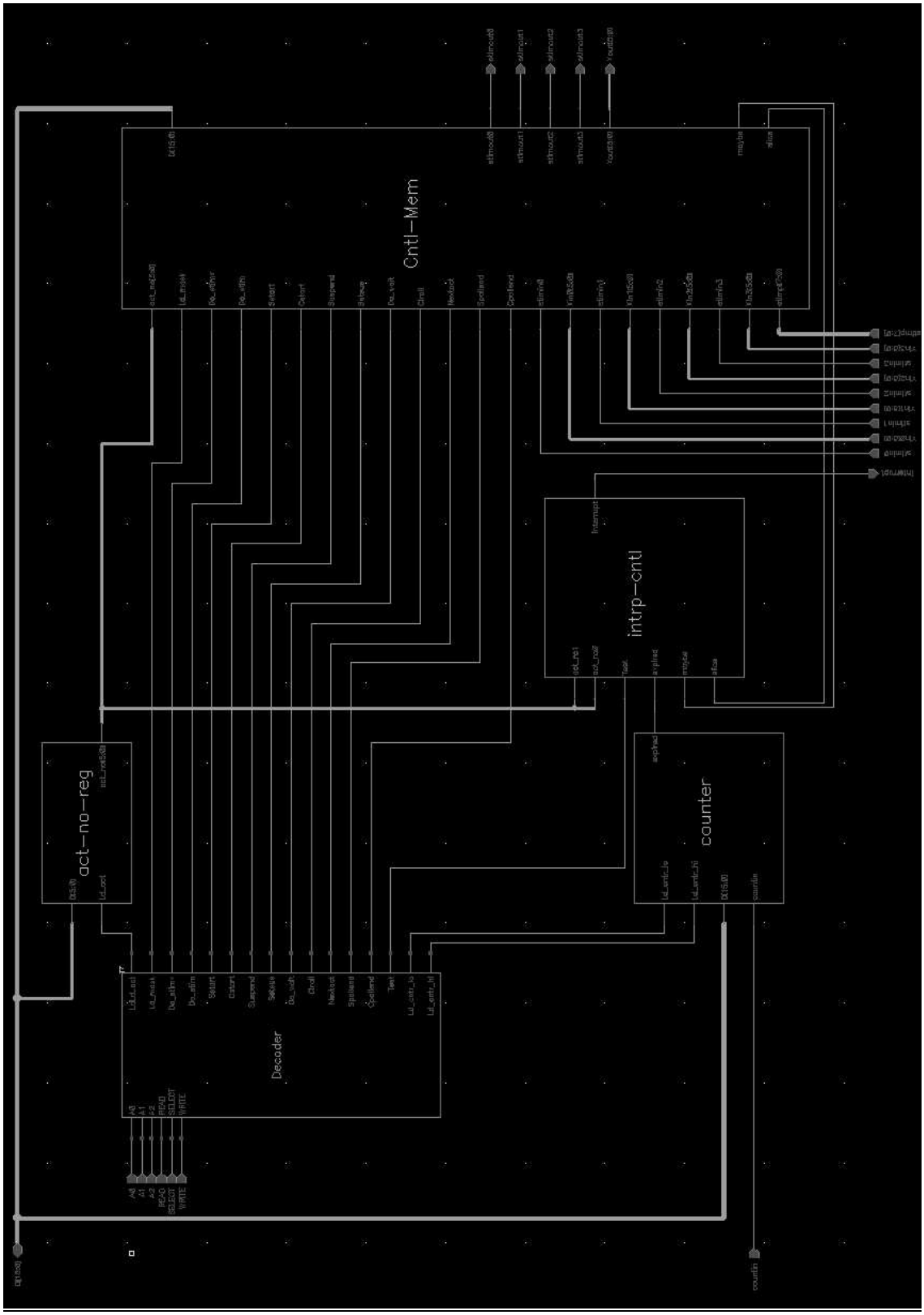



Fig. 4 Block Diagram of original BUTLER

| | Stim0 | Stim1 | Stim2 | .... | .... | Stim13 | Stim14 | Stim15 | Wait0 | Wait1 | Wait2 | .... | .... | Wait13 | Wait14 | Wait15 | Suspended | Started | Polled | Here | Last |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Act0 | 1 | 0 | 0 | .... | .... | 1 | 1 | 1 | 1 | 1 | 0 | .... | .... | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| Act1 | 0 | 1 | 1 | .... | .... | 0 | 0 | 1 | 1 | 1 | 1 | .... | .... | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| Act2 | 0 | 1 | 0 | .... | .... | 1 | 1 | 1 | 1 | 1 | 0 | .... | .... | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| Act3 | 1 | 1 | 1 | .... | .... | 0 | 0 | 0 | 0 | 1 | 1 | .... | .... | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| Act4 | 0 | 1 | 1 | .... | .... | 1 | 1 | 1 | 1 | 1 | 0 | .... | .... | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| Act5 | 0 | 1 | 0 | .... | .... | 1 | 0 | 1 | 1 | 1 | 0 | .... | .... | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| .... | 0 | 1 | 0 | .... | .... | 0 | 1 | 0 | 0 | 1 | 0 | .... | .... | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| .... | 0 | 1 | 0 | .... | .... | 1 | 0 | 1 | 1 | 1 | 1 | .... | .... | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| .... | 0 | 1 | 1 | .... | .... | 0 | 1 | 1 | 1 | 1 | 1 | .... | .... | 0 | . | 0 | 0 | 0 | 0 | 0 | 1 |
| .... | 1 | 1 | 1 | .... | .... | 0 | 1 | 1 | 1 | 1 | 0 | .... | .... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| .... | 1 | 0 | 1 | .... | .... | 0 | 1 | 1 | 1 | 1 | 0 | .... | .... | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| .... | 1 | 0 | 1 | .... | .... | 0 | 0 | 0 | 0 | 0 | 0 | .... | .... | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| Act60 | 1 | 0 | 1 | .... | .... | 0 | 0 | 1 | 1 | 0 | 1 | .... | .... | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| Act61 | 1 | 0 | 1 | .... | .... | 1 | 0 | 1 | 1 | 1 | 1 | .... | .... | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| Act62 | 1 | 1 | 1 | .... | .... | 1 | 1 | 1 | 1 | 1 | 1 | .... | .... | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| Act63 | 1 | 1 | 1 | .... | .... | 1 | 1 | 1 | 1 | 1 | 1 | .... | .... | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

Fig. 5 Control Memory

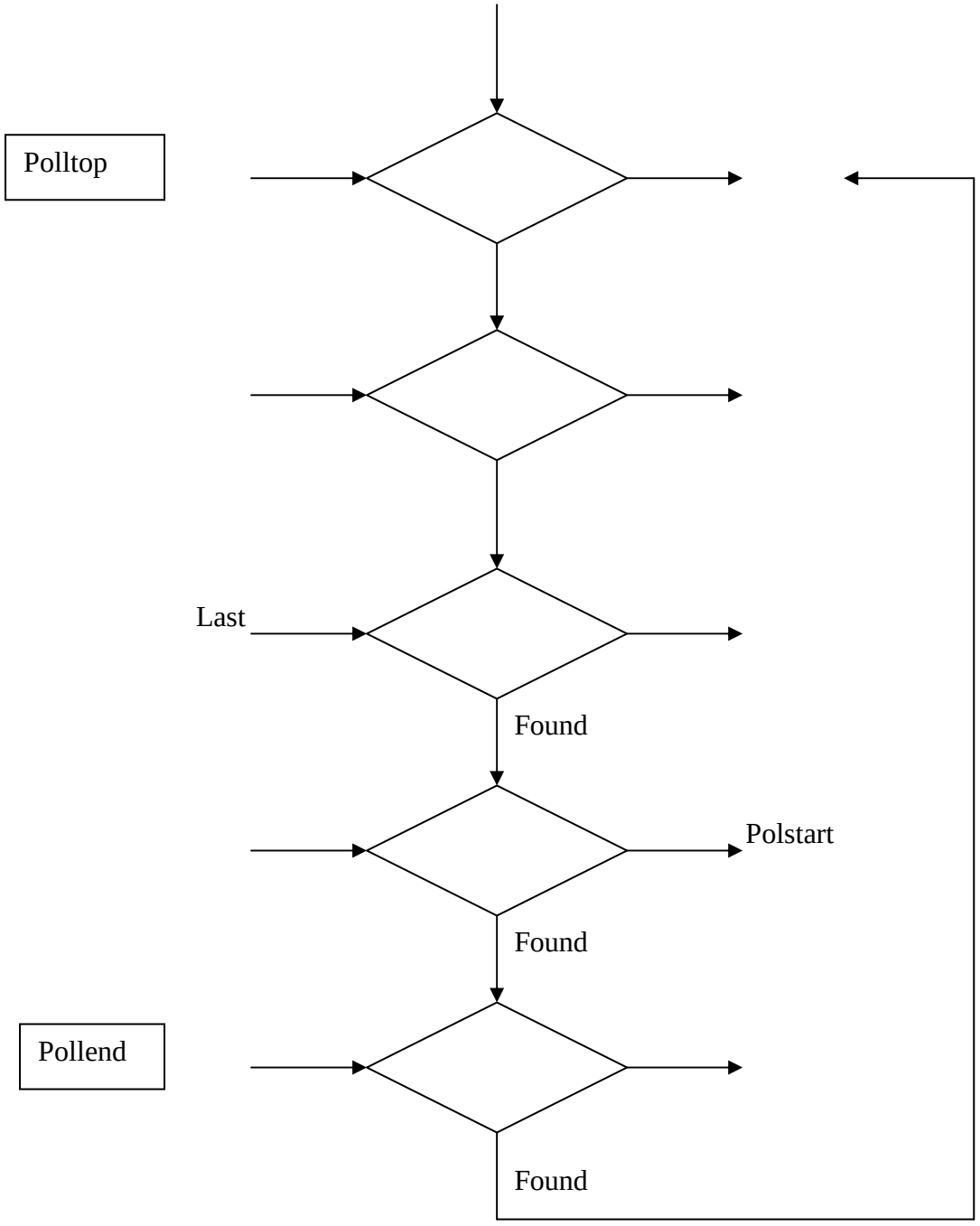



Fig. 6 Polstart Search

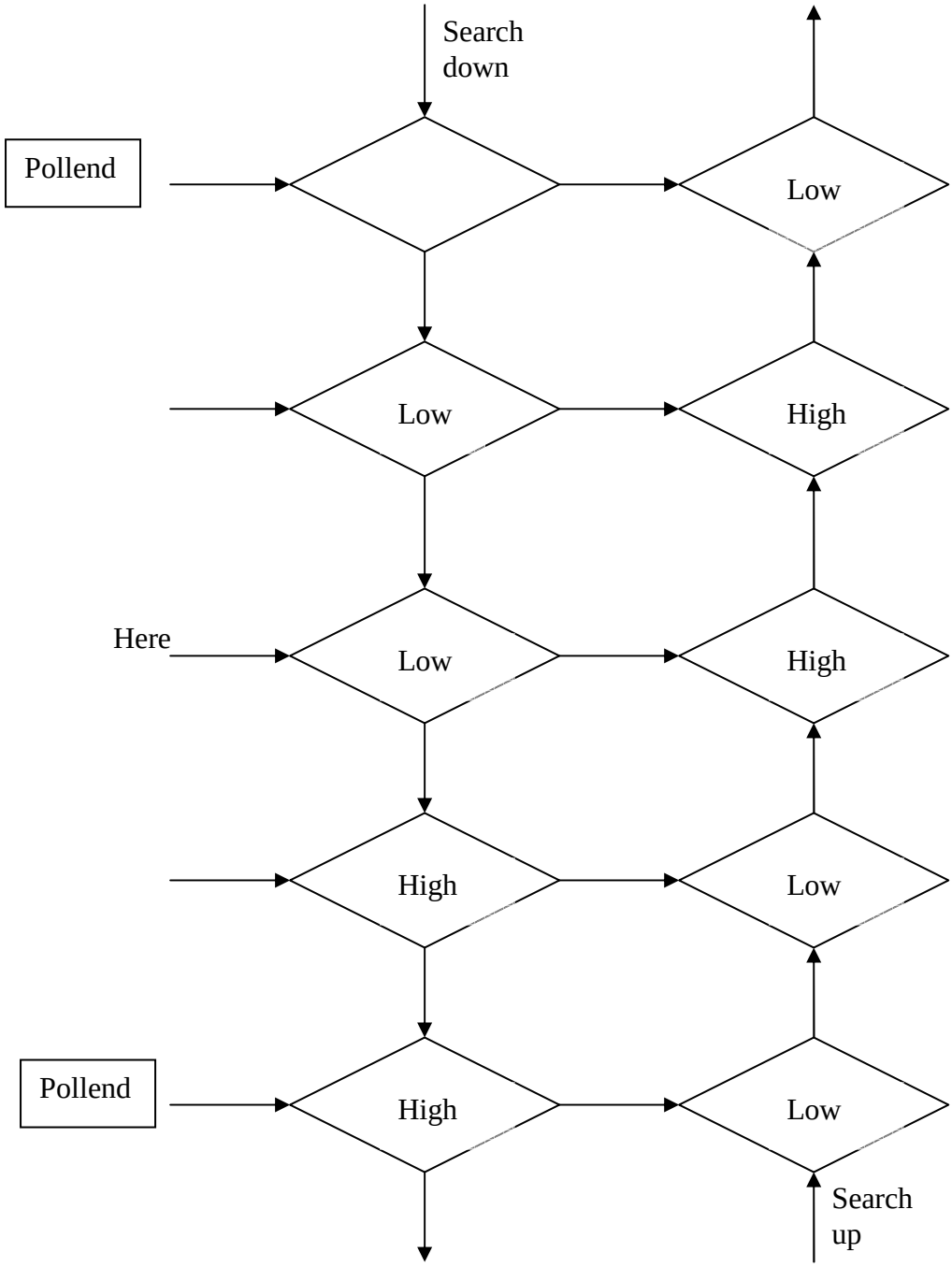



Fig. 7 Search for activities to be included in pollset of Nextact

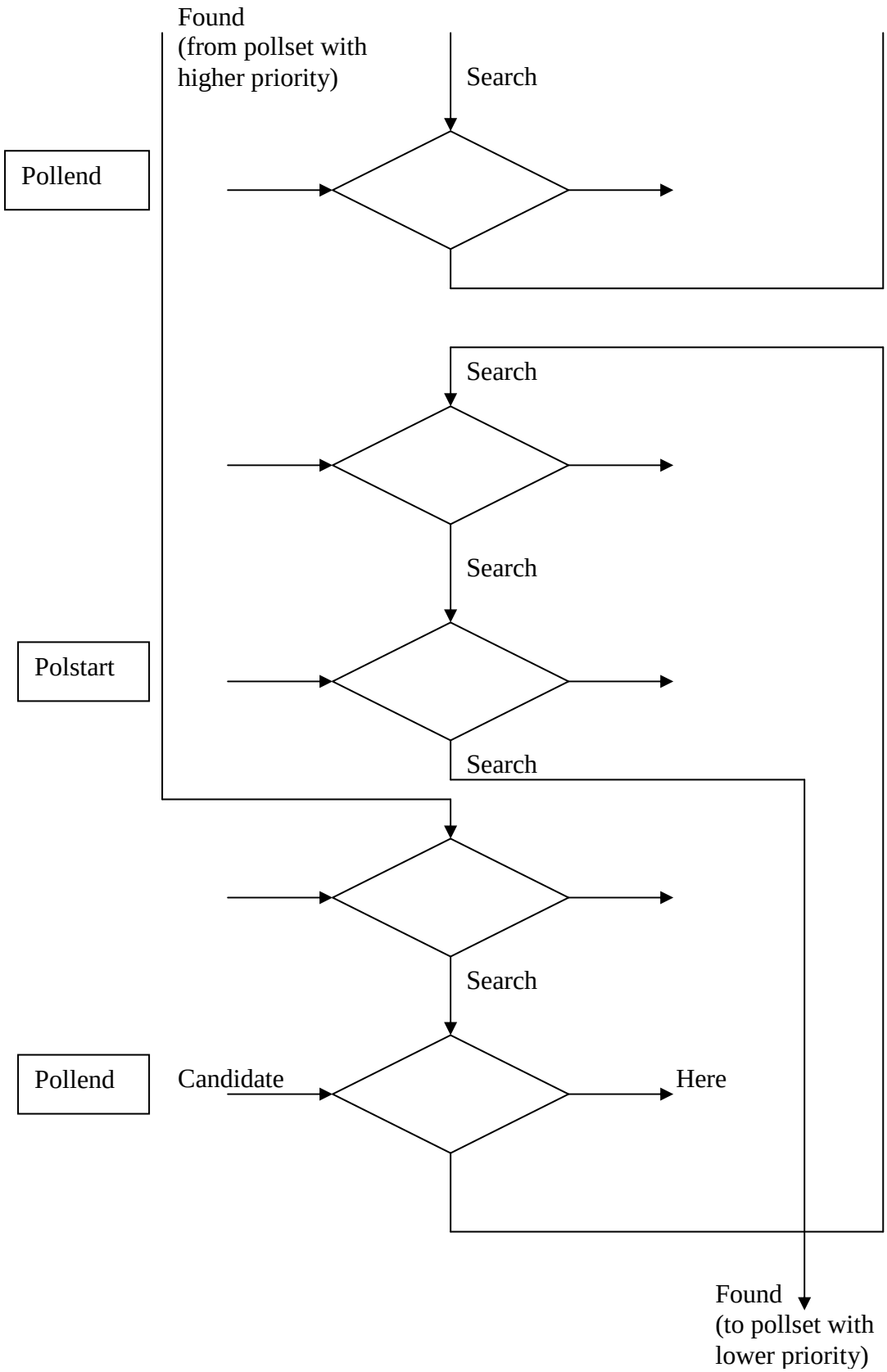



Fig. 8 Round Robin Implementation

| | Stim0 | Stim1 | Stim2 | Stim3 | Stim4 | Stim5 | Stim6 | Stim7 | Wait0 | Wait1 | Wait2 | Wait3 | Wait4 | Wait5 | Wait6 | Wait7 | Suspended | Started | Polled | Candidate | Last |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Act0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| Act1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| Act2 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| Act3 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| Act4 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| Act5 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| Act6 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| Act7 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| Act8 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | | 0 | 0 | 0 | 0 | 0 | 1 |
| Act9 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Act10 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| Act11 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| Act12 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| Act13 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| Act14 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| Act15 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

Fig. 9 Control Memory

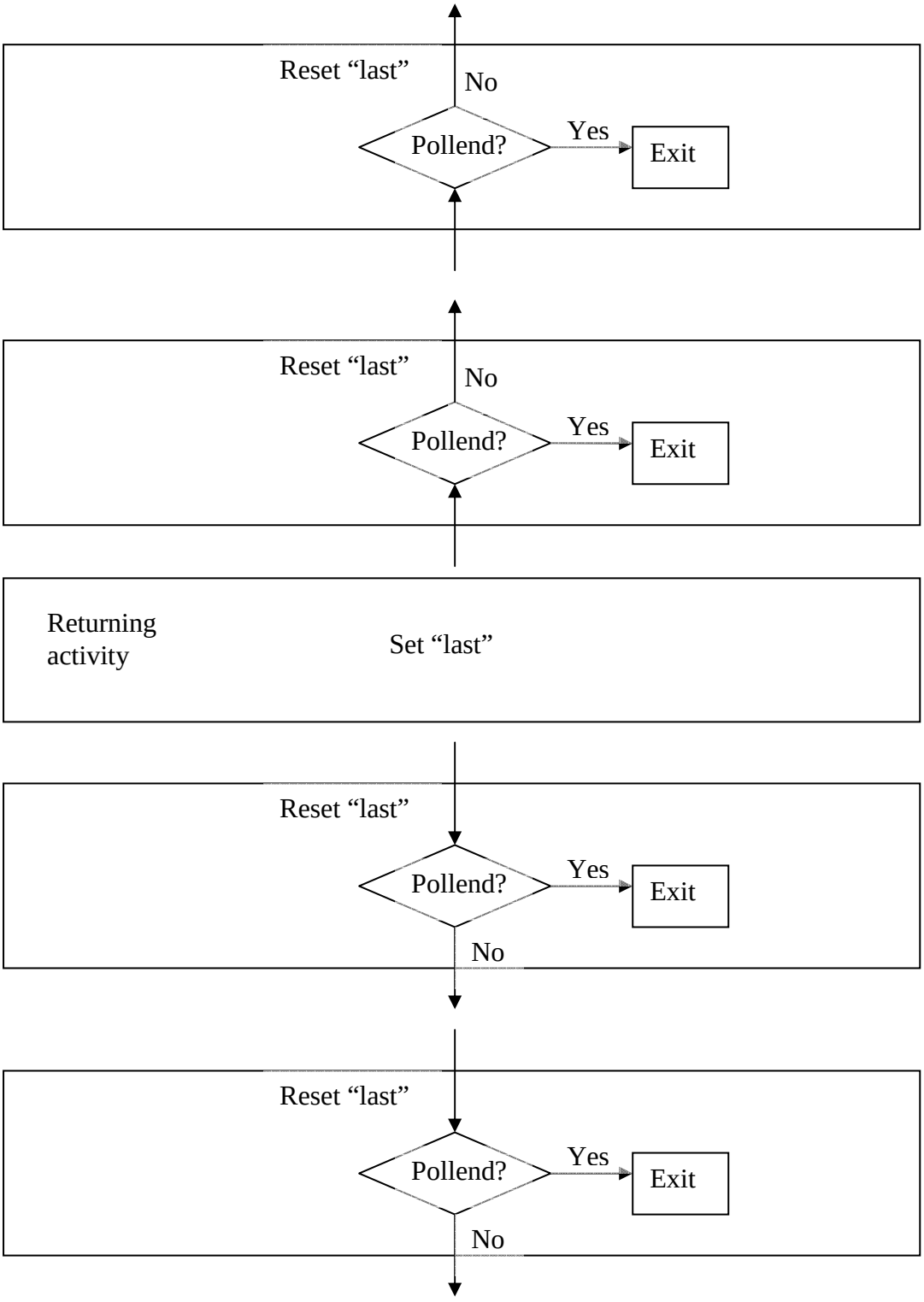



Fig. 10 “last” variable setting logic

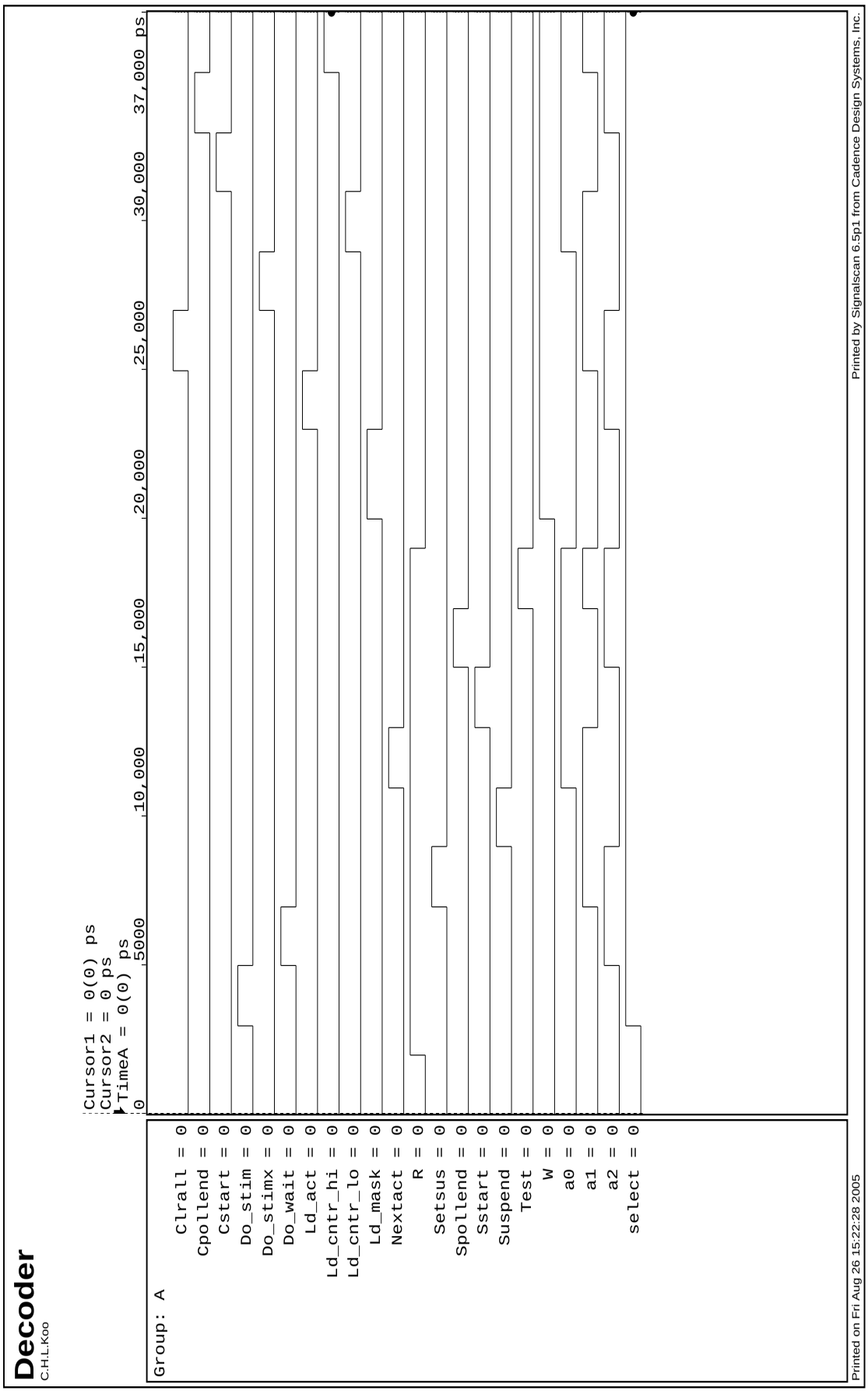



Fig. 11 Signal waveform of decoder simulation

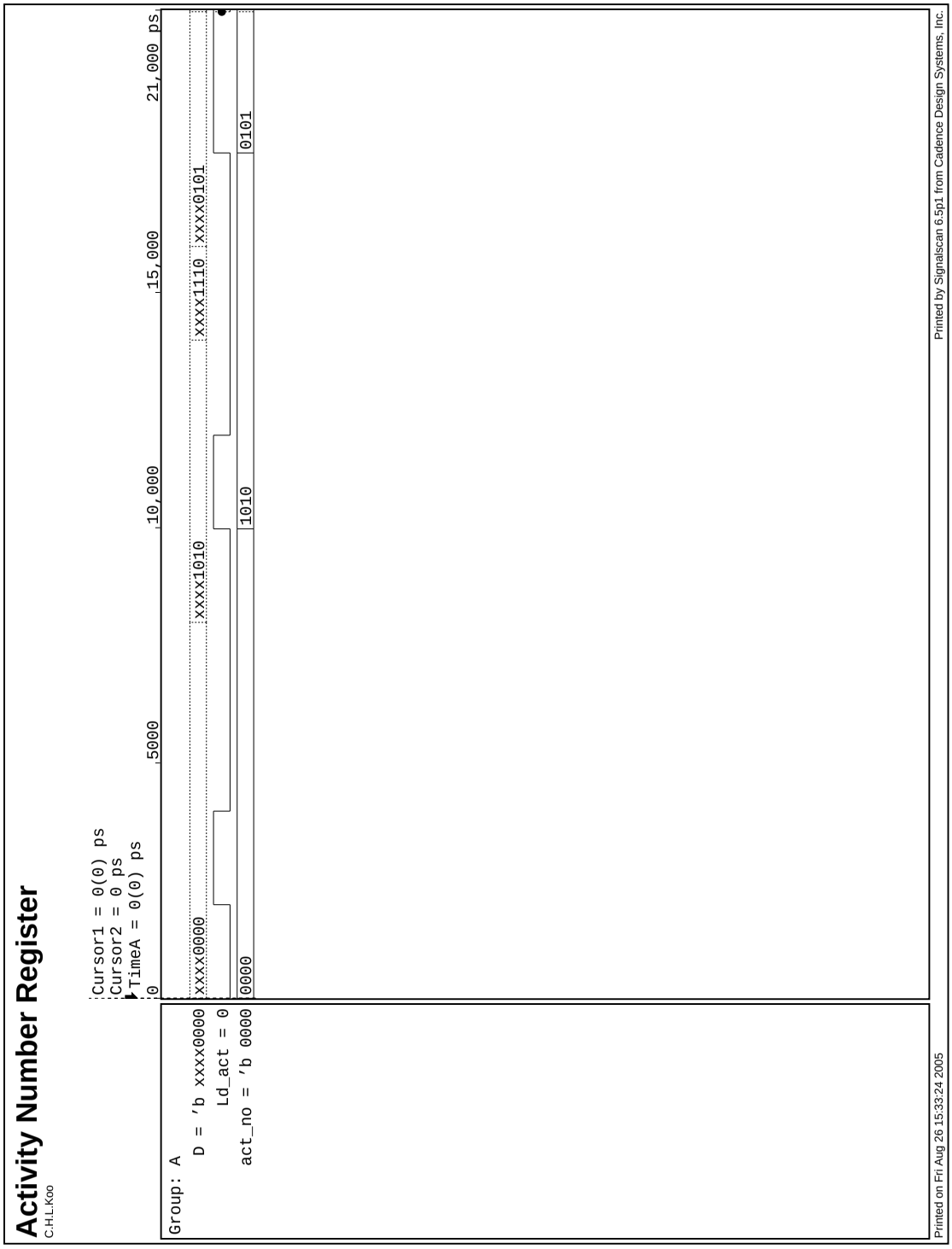



Fig. 12 Signal waveform of activity number register simulation

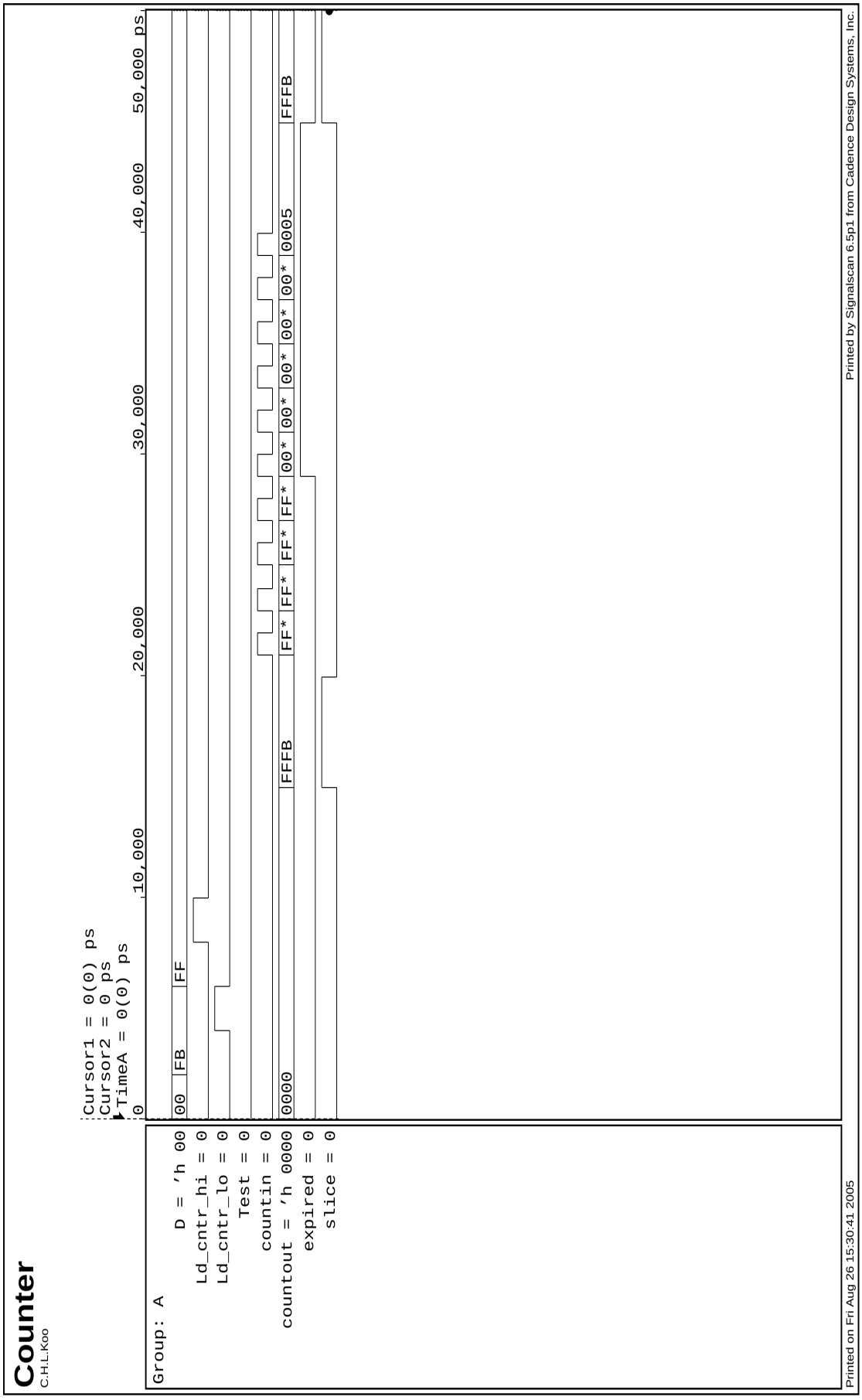



Fig. 13 Signal waveform of counter simulation

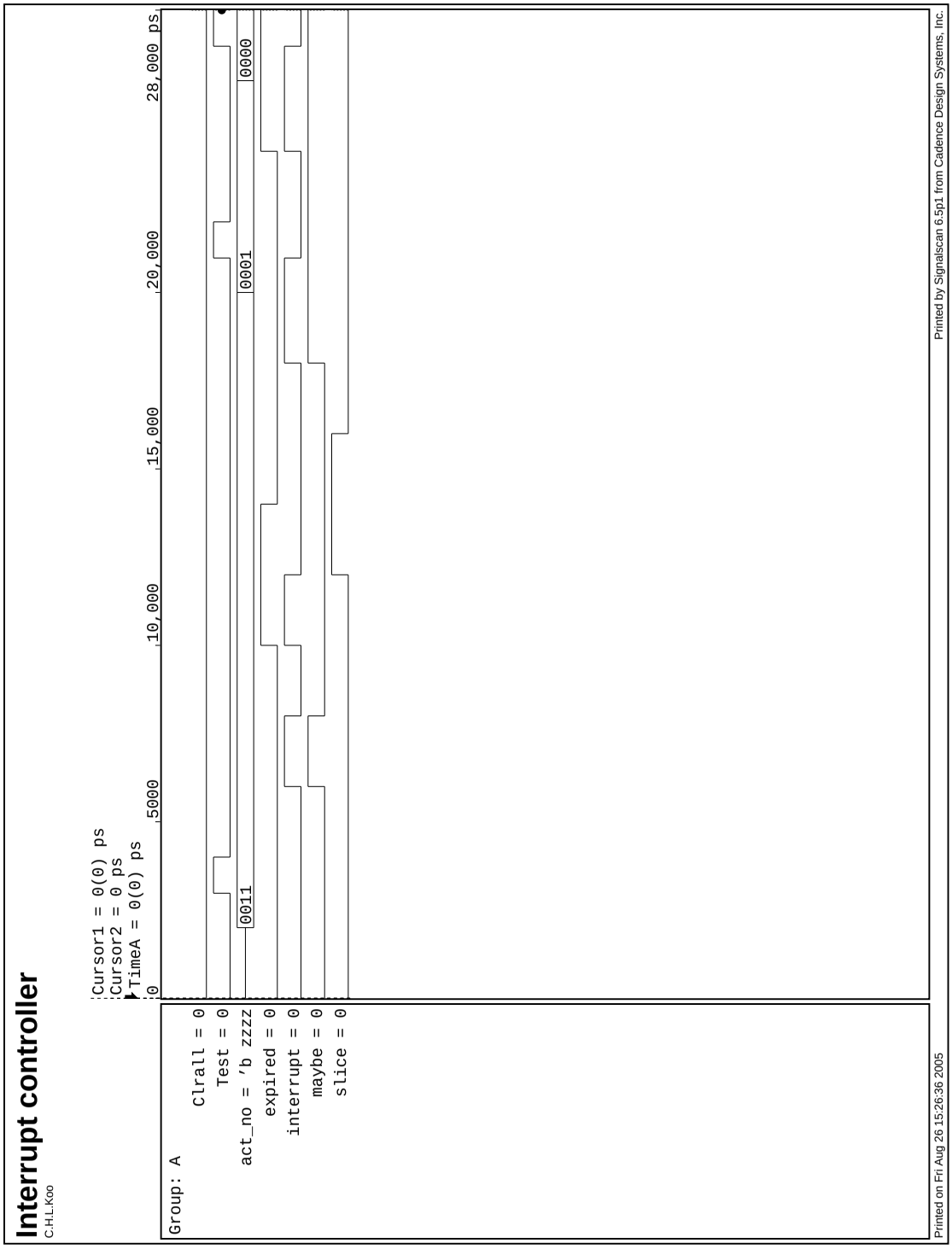



Fig. 14 Signal waveform of interrupt controller simulation

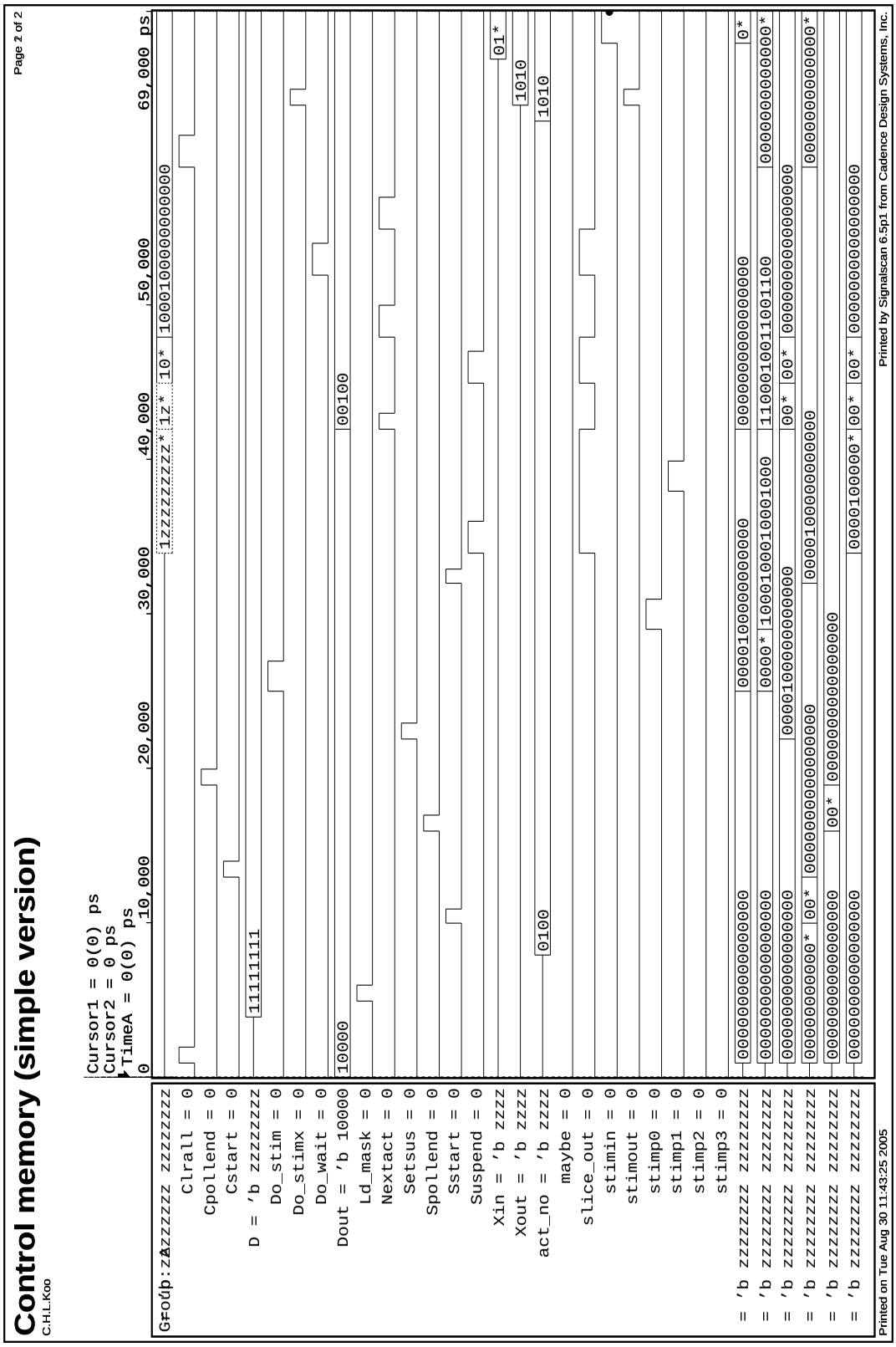



Fig. 15 Signal waveform of control memory simulation (simple version)

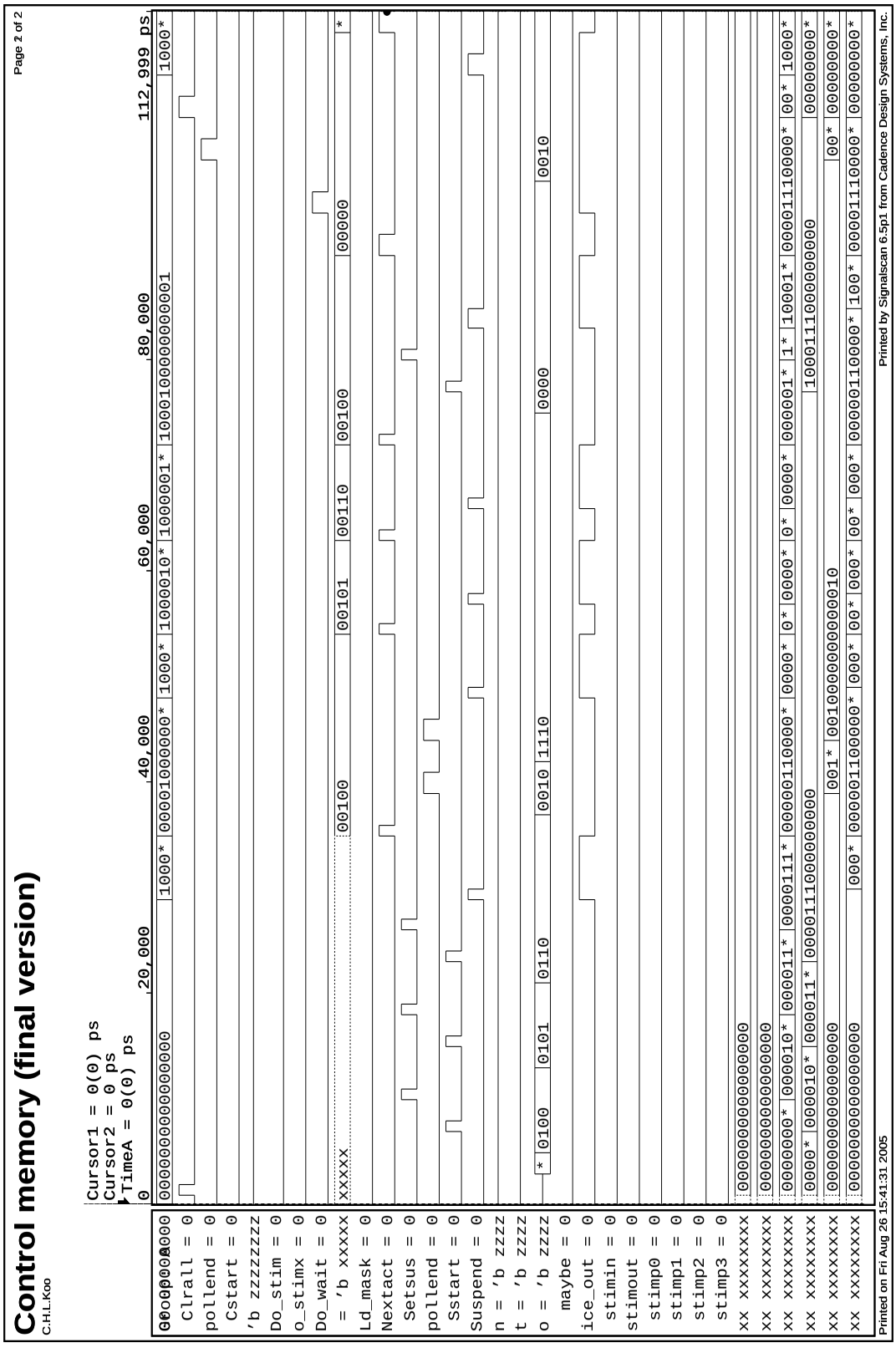



Fig. 16 Signal waveform of control memory simulation (final version)

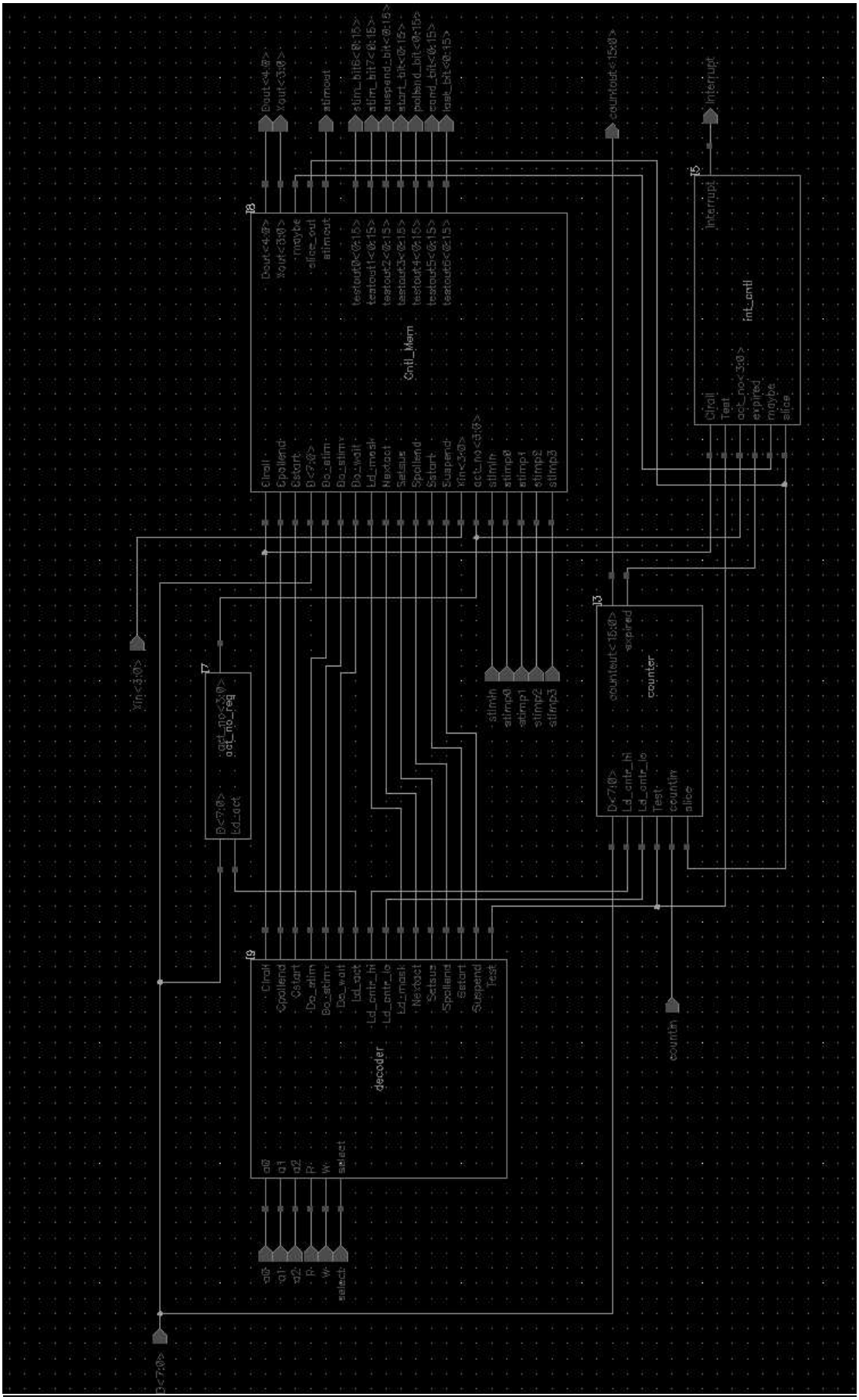


Fig. 17. Block diagram of newly designed BUTLER

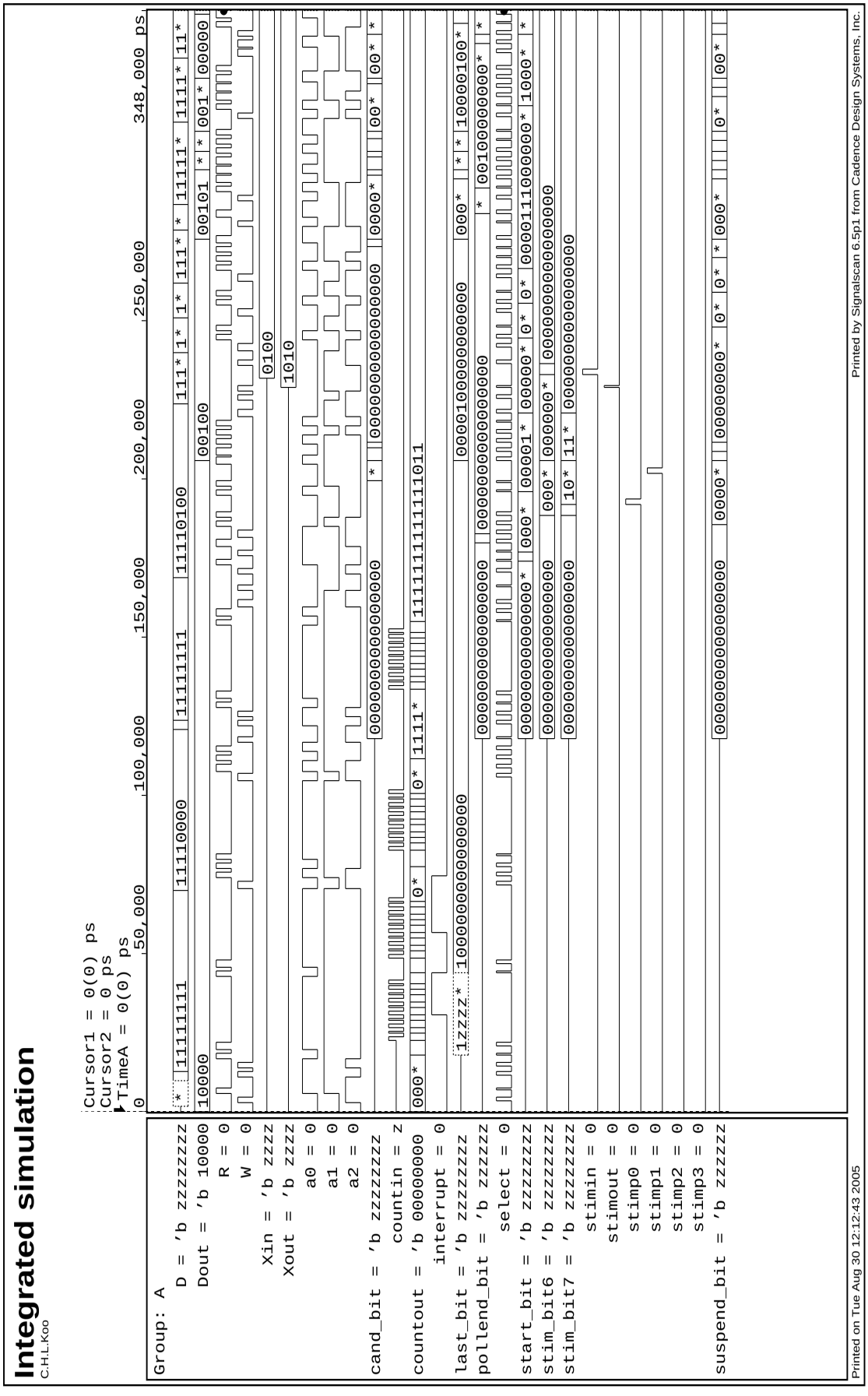



Fig. 18 Signal waveform of integrated simulation