Design memory protection based on embedded operating system with focus on PicoBlaze soft controller

(Full Paper)

Kaishan Huang, Shenzhen University, China
Xi Song*, Greater Bay Area International Institute for Innovation, Shenzhen, China
sissie.song@gmail.com
Chen Jia, University of York, UK, cjmigu@163.com

ABSTRACT

In dealing with inefficient operation system in today’s e-business environment, the paper focuses on memory protection function based on real-time operating system. Originally, RTOS (Real-Time Operating System) lacks the memory protection feature in the basic system design. Since there is no complete memory protection available for operating systems based on PicoBlaze soft controller, the researcher is dedicated to fill the gap. The research explores how existing embedded operating systems realize memory protection and applies processor PicoBlaze to test such function. In addition, the researcher designed memory protection from hardware and software perspectives. The hardware is designed to have additional unit for memory protection while the software is to design assembler code that enables CPU executing related instructions. After testing and experiment, the result has achieved functional objectives and met requirements under restricted conditions of hardware design.

Keywords: memory protection, RTOS, PicoBlaze soft controller, embedded operation system (OS)

*Corresponding author

INTRODUCTION

Contemporarily, electronic business has been faced with various operational challenges over whole supply chains. From a technical aspect, business is increasingly exposed to massive data in virtual environment and largely demanding sufficient operation system to consume data burdens and maintain the efficiency of operation. This paper develops an embedded strategy applied to the operation system, which indirectly controls the hardware. By engaging embedded systems, firms would find it easier for development because such system coordinates programs in different functions to do the same job. Meanwhile, it isolates certain functional programs to reduce couplings among them, enabling easier individual functional modules. Moreover, readability of code in the operation system could be increased, thus leaving more convenience for code maintenance. The operating system could achieve hierarchical management over the code, where the priority of each task and the execution cycle can be expected.

Memory protection is of great significance for embedded systems. Sufficient protection over memories can enhance system reliability and safety. It exactly prevents systems from data crash and is applied to a large range of areas such as life support system and avionics. Traditional memory protection involves logical (virtual) memory and process model programming environments, where each process has a separate virtual address range that maps to physical memory. However, this approach results in extra overhead such as extra memory for page tables and additional performance loss of data transfer between processes. To overcome such limitations, embedded applications offer solutions, by taking advantages of the smallness in size, easiness of use and efficiency of the underlying RTOS (Real-time Operating System). Such simple mechanism can set up boundaries around key data structures and critical threads to prevent accidental access. The operating system can access and control each page of its memory. Since virtualized addresses are superior in terms of security, in that case, OS (operating system) could protect sensitive data by binding data and unauthorized code to code virtual address space. Simply speaking, in a virtual memory system, any applications want to access data when it is running, should have their data and code mapped to the same virtual memory system (Lee & Park, 2017). However, micro-embedded processors are rarely used in virtual memory mainly because of large demands for calculations, which are infeasible in a small processor.

Overall, this project aims at developing memory protection based on a small embedded operating system RTOS (Real-time Operating System). The RTOS acts as backdoors that control the execution of a number of application subtasks and facilitate communications between the subtasks (Predko, 1999). The project adopts PicoBlaze soft-processor from Xilinx to test proposed memory protection method. Inevitably, micro-controller is impossible to implement virtualized memory management since it has neither enough memory space to support the mapping of virtual addresses or computing power to support tremendous data processing. Therefore, the system chooses a hardware memory management module MPU to complete memory protection. MPU assumes the system having a single physical memory space and has nothing to do with the number of process running on the device. The storage space is divided into multiple areas and every process will be assigned a set of zones associated with specific access rights given by the operating system. When the system is running, memory protection unit will check each process constantly to ensure data access complying with its permissions to enforce access rules. This approach saves the overhead of virtualization and simplifies the access control mechanism in the processor.
This paper starts with the literature review to illustrate and analyse how existing embedded operating systems or operating systems of microprocessors achieve memory protection. By reviewing past research, this section provides comprehensive views on how they develop the system and implement the memory protection. Followed by the problem analysis, this section offers more insights about designing this research system. The next section concentrates on design and implementation. Research issues will be provided with solutions, where it considers both software and hardware aspects. Moreover, next section is for result evaluation based on kernel real-time quantitative indicators of the embedded RTOS. The final section will put out the conclusion, conclude the whole research and make prospects for further exploration in related sectors.

LITERATURE REVIEW

Soft-Processor

CPU
Xilinx provides the PicoBlaze microprocessor, an 8-bits soft-processor which can be run on the FPGA like Spartan-3, Virtex-II and Virtex-II PRO (Chapmen, 2011). As Figure 1 shows, the internal structure of the KCPSM3 (PicoBlaze) can occupy 96 slices on Spartan-3 board. A single block RAM can be used to store 1024 instructions for a program. Theoretically, the performance is expected at about 43 to 66 million instructions per second at 87MHZ frequency. Performance varies due to the device type. By comparison, the internal structure of ARM7 is more complicated than that of PicoBlaze. The IPS of ARM7 is 40 MIPS at 45 MHZ, which is less than its counterpart indicating that PicoBlaze is a well-performed micro-controller.

PicoBlaze is in RISC architecture (Reduced instruction set computer). It contains 16 registers and a stack and could store 31 return addresses. Apart from that, PicoBlaze has a bank of scratchpad memory with 64 addresses. Moreover, PicoBlaze has a strong expansion and is easy to connect 256 I/O ports to hardware. Considering the objective of this project with regards to memory protection design, more insights of the memory architecture of PicoBlaze are required.

As is shown in figure 2, PicoBlaze uses the Harvard memory architecture, which is characterized by physical separation of memory instructions and data. Therefore, data cannot be stored in instruction memory (Freeman, 2005). Since each instruction takes two clock cycles, as mentioned in introduction, MPU will be added to protect memory.
Figure 3. PicoBlaze internal structure (Chapmen, 2011)

Figure 3 exhibits the internal architecture of PicoBlaze. It pictorially shows how the address links each unit inside. KCPSM3 supports a program that is up to a length of 1024 instructions and uses one block memory. The length of data is 256. The ALU (Arithmetic Logic Unit) provides simple operations in processing unit. There are basic logical operations like ADD and SUB and others such as ADDCY and SUBCY that are more than 8-bits operations with carry flag.

Interrupt is a very important signal in KCPSM3 and takes an irreplaceable role in system design. PicoBlaze calls the instruction at address 0x3FF from which users can define jump vector to ISR (Interrupt Service Routine). At this time, INTERRUPT_ACK will generate a pulse on the output. In the final design, the system will trigger the interrupt service routine based on the interrupt signal to handle tasks of the highest priority.

Figure 4. Waveform of interrupt signal (Chapmen, 2011)

Figure 4 presents the waveform of the whole interrupt operations. When delay loop begins, it only repeats the address 005 and 006. Before calling the instruction at 3FF address, there is a need for clock cycles to respond. After defining a jump vector, the interrupt service routine follows. At the same time, the INTERRUPT_ACK signal increase in the waveform, showing that it captures of interrupt signal.

Programming tools

Overall, the programming is divided into software and hardware. The software is OpenPICIDE, an assembler development environment for soft-core processors provided by Xilinx (Fautronix.com, 2018). It includes assembler, compile and simulator used for debugging code. The main function of OpenPICIDE is used to make instruction memory to be read by PicoBlaze.
PicoBlaze requires the instruction memory since it lacks instruction store as a VHDL module. The assembler can produce ROM files added into the system.

The hardware is also provided by Xilinx namely ISE. Since embedded feature of ISE meets project requirements. MPU of the hardware part is compiled via ISE. Moreover, ISE provides ISim, a simulator that can check hardware error and does not need to flash system into FPGA board. Additionally, it is able to view the value of the signal at every clock cycle (Xilinx.com, 2011).

**Embedded Operation System**
Comparing to general operating systems, embedded OS is featured by initiatives to keep things simple and efficient. To specify, there are few software layers between user and hardware. Meanwhile, it supports a limited range of hardware device. The basic structure of embedded OS is provided by Dr. Mike, namely RTOS (Real-time Operating System). RTOS can accept and process at a fast speed whenever external events or data are generated. The processing result can control the production process or respond quickly to the processing system within limited time. There are three representative features: 1) high-precision timing system (Xiaodan & Hongbin, 2007); 2) multi-level interrupt mechanism; 3) real-time scheduling mechanism (Cho et al., 2006).

As for data structure, OS is expected to achieve core functionality that is to store vital information to support operating the system. Task structure is very important when a process is created.

<table>
<thead>
<tr>
<th>Task Control Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
<tr>
<td>DELAY</td>
</tr>
<tr>
<td>LAST</td>
</tr>
<tr>
<td>PRIORITY</td>
</tr>
<tr>
<td>PC</td>
</tr>
<tr>
<td>DATA</td>
</tr>
</tbody>
</table>

Figure 5. TCB block diagram

Figure 5 shows the TCB structure of a task. Each task will be assigned a priority level so as to determine what task should be executed then. The data structure is stored in memory. Information of task status indicates its execution. PC and DATA are necessary in hardware design, in which PC refers to instruction address while data stands for data address.

The tasks state can be divided into three order lists as shown in figure 6: the ready list, running list and waiting list. Multitasking execution can support several tasks running in the system. However, only one task can be running at a time while other tasks should be prepared to run in the ready list or wait for an event in the waiting list. In hardware design, MPU (Memory Protection Unit) requires recognizing which task is running.

When the system starts executing, the first step is to construct the task and followed by starting interrupt timer in order to trigger the context switch. In addition, context switching is used to schedule different tasks to run on the processor. The
switching order between tasks is based on the priority scheduling algorithm. Moreover, priority of the tasks can be set in advance to determine the switching order. In a general sense, two task queues are required to run the task. Task with the highest priority is right out of a queue. Another queue is used for blocked tasks.

**Existing Memory Protection**

**Hardware support for memory protection**

Achieving memory protection with the support of hardware has undergone evolutionary processes. During early years of system protection, memory protection is completely implemented by software without any hardware support. Since mid-1960s, more focus and importance have been attached to hardware support. Initially, hardware approach refers to adding certain easy-to-implement instructions to regular computer instruction set. These instructions are used to transfer functions from the functional segment to the function registers (Wilkes, 1982).

Later, since the computer has a fairly large microprogram memory, new design is proposed as security kernel should be included in the OS design. Though all security-related features are concentrated in the kernel, in order to enable the rest of the operating system having the same protection mechanism as security kernel, it can choose to construct partly of the kernel. Other parts of security kernel need permissions to access.

Additionally, cost issues are worth mentioning. In general, cost is reflected in the extra hardware particularly when extra hardware slows down the running speed and takes up extra memory. Managing costs to a reasonable ratio is a longstanding problem to be solved.

**Memory protection system in tiny embedded processors**

The system is based on 8-bit microcontrollers, which is similar to Picoblaze. The architecture of tiny processor is very simple, where software modules running on the processor can access the entire memory through a single address bus. Due to resource constraints of processor, the address space could not be divided. To deal with that, the system relies on memory map as a data structure. It can record ownership or layout information of whole address space. The address space of each process is allocated into contiguous segments and each segment is partitioned to a software module.

In addition, software modules are implemented by run-time checks, which can ensure all memory accesses are limited in segment that partitioned to module. However, these are implemented through rewriting compiled binary, which will increase performance overheads in software. To overcome such limitation, the system promotes the implementation of store, call and return instructions in controller to realize run-time check in hardware.

For system structure, memory protection is normally achieved with software/hardware co-design. Such approach eliminates the cost in architecture extensions. Overall, the system is mainly implemented in software, where the memory map can be divided into three parts as protection domains, data structure and memory map checker. The protected domain is created in the data storage address space of embedded processor. Each module stores its status in its own protection domain. Modules are limited to write memory out of their domains while the system is running. Limited memory does not support statically dividing address space into multiple domains. Instead, the address space of the microcontroller is divided into blocks of the same size. Memory map contains access to each address space block and two kinds of information. The first is ownership information for each memory block while the second is encoding information about the memory layout (Kumar et al., 2007).

MMC (Memory Map Checker) is vital in checking the access of software components. Its main function is to enforce the protection model described earlier. Program can only be written to its own domain. As figure 7 shows:

![Figure 7. MMC block diagram (Kumar et al., 2007)](image)

MMC works as a unit that intercepts the signal generated by CPU which is used to write to data memory. If the written address is valid, MMC rewrites to the data memory. Such design meets the requirements of this project design. It does not involve additional computing requirements for the processor or take up too much memory. It indeed implements memory protection for RTOS built by PicoBlaze, thus avoiding excessive system resources. For the system itself, those ways designed for memory protection succeed in reaching a trade-off between objectives and resource constraints.

**Memory management to support multitasking**
The memory management is designed for an FPGA (Field-programmable Gate Arrays), which is based on a computer system. Differing from computers, FPGA fails to support physical MMU (Memory Management Unit) of the microprocessor. To compensate that, MMU can be designed as a module running on the system to handle memory management. Similar to other computer systems, various tasks of this system will share the memory as a resource. However, it is impossible to predict which tasks will be combined and executed in actual system operation. Thus, MMU is expected to support virtual addressing.

Moreover, program storage physically separates from data storage. Data memory of a task can be made up by task registers scattered on the FPGA. FPGA computer can run several tasks at the same time with multiple processors. When multiple tasks accessing the same memory bank, tasks may easily come into conflicts. Under such conditions, an effective mechanism is expected to ensure tasks not affecting each other. In general, a system architecture will have several memory banks for parallel access. The existence of the memory bank is generally flexible, either internal RAM or external RAM. Since external RAM is larger, MMU will manage them in a unified way.

To make data memory addressing of a task independent from other tasks at a time, virtual address is necessary for that. Each task will get an independent memory space mapping in virtual address and begin at 0X00. The virtual address of every task will be translated to physical address. In order to achieve that, MMU needs to store information to enable each task mapping virtual address to physical address. The memory space of RAM will be separated into fixed page number in the same size. Each task memory can be assigned several pages. Page and task affiliation will be stored in MMU.

![Figure 8. Page table block diagram (Danne, 2000)](image)

Figure 8 as a mapping table shows the correspondence between pages and tasks. Task will use virtual address when it visits memory. The upper part of virtual address can process page table according to task ID. The output of page table will be added into lower part of virtual address (Danne, 2000). This is aimed to get physical address. MMU can check and recognize whether a task tries to access an address beyond its range of address space. The X mark as shown in the figure indicates that this page does not belong to any task. If the address is invalid, writing access will be denied while reading access will return a value of 0.

Such design implements memory protection as a good example. The FPGA computer system has enough memory to support mapping of virtual addresses. MMU stores mapping information of tasks and checks address mapping. However, they are not the case for this project. PicoBlaze fails to provide enough memory space to support building virtual memory. As an 8-bit microprocessor, the available memory resource is relatively rare. It can only provide a small amount of memory to the task that is far from memory capacity required for virtual memory mapping. Briefly speaking, PicoBlaze is a very small core used for building small systems. Therefore, such design only achieves certain expansion in memory protection.

**Memory protection in RTOS**

The RTOS is developed for embedded system. The micro-kernel architecture includes process/thread management, which is essential for memory protection, real-time scheduling, communication, timer, and interrupt handler. Memory protection is implemented by memory management software supported by MMU. This system adopts virtual memory concepts, in which 3-level paging is used for memory protection. Besides, memory protection requires MMU and memory management tables, which consists of page tables or segment tables. However, the page size that microprocessor supports is 4-8KB bytes. Such size is too big for a small-embedded system, thus requiring to be reduced. For any small-embedded systems, more efficient memory usage and less memory requirements are very important. Therefore, it is vital to share thread stacks among processes. Additionally, enlarging or shrinking data dynamically based on requirements also promotes the efficiency of memory usage (Suzuki & Shin, 1997).

The memory protection method is generally based on the processing of the address. Memory address is based on physical address or logical address. In general, there are five main types of existing memory protection methods. The first one is Bit Map, which divides physical memory into pages to realize memory protection. In such method, each process needs check the protection information when it accesses the corresponding page. The second approach is segmentation. It splits logical address into multiple segments while each segment will be mapped into physical memory space. This method also needs a memory management table that includes STES (Segment Table Entries) to manage segments.
The third type is multi-level paging. Figure 9 above shows the structure of multi-level paging as a 3-level structure of page table. Comparing with segmentation method, multi-level paging divides logical address into pages of the same size. All pages will be mapped into physical address space. Such way facilitates reducing the size of page tables by hierarchical layer according to different levels.

The fourth method is paged segmentation. Each segment of logical address space is divided into several pages and all pages should be mapped into physical address space. In practice, it uses several page tables and a segment table to manage memory. The fifth approach is short-circuited segment tree. Comparing to multi-level paging, it just uses first page entries, which reduces the overhead processing and total page size. Following figure 10 is the structure of Short-circuit segment tree. Though it presents five levels of pages, it is apparent to find that the first level page table is directly connected to the bottom page table. The second to the fourth-level page table of stack area and data are skipped. This is actually aimed to ignore mid-level pages so that it can reduce processing overheads caused by extra table size.

This new memory protection method for small-embedded systems based on modifications on the fifth method. Similarly, this method also skips the intermediate-level page table. The difference is that it enables a PTE (Page Table Entry) in every level page table to map to the physical address space and skip page tables that only one PTE is used (Suzuki & Shin, 1997). This method ignores intermediate-level page tables and uses larger size pages which makes it possible to reduce processing overheads and total table size.
Memory Protection for embedded OS Kernels

OS is developed for embedded OS kernel based on three abstractions. The first one is process as basic unit of execution. The second one refers to files that include a lot of I/O operations. The last one is sockets as implementation of Internet protocols which are available by a socket interface. These three abstractions have an API (Application Programming Interface), which is exported by kernel system call interface.

The design is portable among CPU architecture that provide page-level protection with MMU hardware. The core idea is using page-based memory protection to prevent one process accessing to the memory area of other different processes. In addition, it also prevents user processing access kernel memory. The virtual address is one-to-one mapped to physical address, making all processes assigned with same logical address and physical address. Concerning memory management, a one-to-one mapping greatly simplifies the design of the memory management subsystem.

Moreover, the protection mechanism is based on domain, which consists of memory pages that are mapped with a set of page tables. Apart from that, memory pages related to the kernel are combined into a set called kernel map. Kernel map is mapped to all domains. However, it can only be accessed in special mode (Miler, 2002).

Overall, there are three basic memory management data structures supporting memory protection: region table, page table and kernel map. As defined, region table is an array used for tracking the physical memory. Page table defines protection domain that contains relevant process. Kernel map refers to a list containing all page tables, which can only be accessed by CPU in special mode. These three data structures are used in conjunction through kernel memory management routines.

Furthermore, the basic unit of memory allocation is region. Region is substantially a page alignment and a sequence of consecutive addressable locations tracked with a starting address. In addition, the length should be a multiple of the page size.

![Region Diagram](image1)

Figure 11. Memory allocation unit block diagram (Miler, 2002)

Figure above exhibits the structure of memory allocation unit. The data structure of region is used to track the allocation of memory. Besides, every region is tracked by its starting address (Start field as shown in Figure 11) and length (Len field as shown in Figure 1). Each region belongs to the process, which is assigned originally, or the ownership has been shifted after allocation. The Proc field is used to track the region belongs to which process currently. Two double-linked lists are maintained with Prev and Next field. The former is free list that shows regions not assigned to any processes. The later is allocated list, which presents regions already allocated to certain processes.

![Page Table Record Diagram](image2)

Figure 12. Page table record data (Miler, 2002)

Figure 12 presents page table record data structure. Kernel can track page tables efficiently with these records in next field. The refcnt field will track total number of threads within the domain when multiple threads are executed through a single protection domain. Moreover, the pd field indicates actual page table data structure. This design suggests that page tables are set up contiguously in memory.
In this system, memory is used to hold kernels. The associated data structures of kernel are mapped into all page tables in this system. Moreover, these pages can only be accessed when CPU is run in special mode. Generally, such model will not occur unless there are interruptions or system calls. Briefly speaking, such protection approach is to deny any other forms of memory access to protect kernel's memory.

The kernel mapping is an array of kernel map entries, among which each entry represents a region where it entered in kernel map. Figure 13 shows the structure of kernel map entries. To specify, the Start and Len fields are used to track the starting address and length of the region. The Attr field is used to store attributes associated with pages.

Such design is based on kernel security to implement memory protection, which is suitable for embedded systems. Comparing with traditional memory protection methods, it simplifies code size in address translation and is applicable to different processors.

Figure 14 shows the structure of intermediate-level skip multi-size paging. It skips the stack area from the second to fourth levels while the first level of the page table is directly connected to the fifth level. As mentioned before, virtual address and physical address are mapped through page tables, thus helping achieve memory protection.

This example implements memory protection in a real-time operating system developed for small embedded system. Taken features of small-embedded system into account, such method will enhance memory usage and reduce processing overheads within the scope of system conditions. This method is suitable for this system by modifying in some sections compared to current memory protection method.

**PROBLEM ANALYSIS**

**Operation System Features**

**Users and files management**

Unlike traditional operating systems, this system is run by assembler language and does not require user interaction. Therefore, there is no need to grant user permissions to start a new process. Entering data from the I/O side is the only form of input,
which is generally not entered by user of the device. Such data does not affect execution of programs in the system. So, it does not require user functionality to control system.

Similarly, the file system is of little necessity because operating system can get inputs from sensors to have real-time controls and clear the read data. Since there are no users or files, there is no temporary data to manage. Therefore, building additional management system is unnecessary.

Comparing to other operating systems, there are no complicated interaction and management system. Some common basic security functions are difficult to implement in this project. Basically, it could not get involved in tasks, which are running, leading to a less sufficient error handling. More importantly, it is expected to ensure other tasks working properly when context switches. When the system is in an unknown state, it is suggested to restart.

**Concurrency of tasks**
As a basic function, processes should be able to run concurrently. Moreover, it is necessary for the system to run tasks at the same time. When the context switches, other operations can be performed. Apart from that, it must also have the functionality as a scheduler that ensures all tasks could run.

Since the key point of this project is to implement memory protection, each task should be able to allocate the required memory and protect the allocated memory when performing memory management. The purpose is to avoid interferences between tasks. Particularly, in facing with weakened error handling mechanism, memory protection plays a critical role in ensuring the system running properly.

**System Architecture Design**
Considering the limitations of PicoBlaze, basically, the system is expected to avoid making extra overhead. In order to complete the project, the hardware should be modified. At present, adding hardware extensions based on the original system is the least influential on system overheads. As aforementioned in the second chapter, the ultimate design is to build a unit called MPU (Memory Protection Unit) to implement the memory protection. The motive for adopting such design is that PicoBlaze does not have to change itself while it is within the capabilities that it could withstand. The design makes reasonable use of the processor's expansion capabilities, which is the least expensive design method at present.

Moreover, the project is exclusive from other memory protection methods like virtual address mapping for two reasons. On one hand, the system is a small-embedded OS, which could provide enough memory to build page tables. On the other hand, there is insufficient computing power to support calculation of virtual maps. PicoBlaze is a micro-controller that is not capable in massive operations.

Furthermore, due to the expansion of the hardware, the extended hardware should be tested separately to ensure that it could run independently. Followed by testing the whole system, problems could be discovered and processed in a timely manner.

**Estimation Of The Result**
In the first place, there should be absolute isolation of data memory if the memory protection unit finds that the task access is out of the range beyond its assigned address segment. It should block the access to RAM. In addition, monitoring and protection of shared memory is implemented during inter-task communication to prevent concurrency issues. Under these conditions, this project is expected to be successful. On the contrary, the project is likely to fail if memory protection module does not block the wrong task to write data into RAM in time, or block access to data memory after data have been written into RAM.

<table>
<thead>
<tr>
<th>F1</th>
<th>Control on/off</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2</td>
<td>Support more than 8 tasks to run</td>
</tr>
<tr>
<td>F3</td>
<td>Set address range with base and limit registers</td>
</tr>
<tr>
<td>F4</td>
<td>Set read/write authority for address segment</td>
</tr>
<tr>
<td>F5</td>
<td>Set address segment for 8 tasks to make comparison</td>
</tr>
<tr>
<td>F6</td>
<td>Check task address and block cross-border address access</td>
</tr>
<tr>
<td>F7</td>
<td>Give an interrupt signal after a block</td>
</tr>
<tr>
<td>F8</td>
<td>Protect RAM (data memory)</td>
</tr>
<tr>
<td>F9</td>
<td>Protect ROM (instruction memory)</td>
</tr>
</tbody>
</table>

In addition, MPU should implement above features. Moreover, all tests are list as follow, covering full functions implemented by MPU.

<table>
<thead>
<tr>
<th>Table 1. Features of MPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 2. Functions of MPU</td>
</tr>
<tr>
<td>Name</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>Test 1</td>
</tr>
<tr>
<td>Test 2</td>
</tr>
<tr>
<td>Test 3</td>
</tr>
<tr>
<td>Test 4</td>
</tr>
</tbody>
</table>

**DESIGN AND IMPLEMENTATION**

**Hardware Design**

This research is dedicated to design feasible solution to implement memory protection. As discussed before, there are several design choices. The first one is to design an external hardware called MPU (Memory Protection Unit), of which the structure is simple, and the cost is low. Such design will not take up too much memory resource. The second method is to use virtual memory technique, which is suitable for large operating systems since they can schedule enough memory resource to store address information and support address mapping. The last approach is to establish memory protection mechanism based on kernel, which indeed uses the technology of virtual memory.

Concerning this project, the RTOS is designed as a small-embedded operating system while PicoBlaze is an 8-bit microprocessor. So, it is impossible to implement memory protection with virtual memory technology because of such limited memory resource. On this premise, the project designs MPU to realize memory protection.

Figure 15. MPU block diagram1

Figure 15 presents a simple structure of MPU, where CPU PicoBlaze could not write data to the ROM (instruction memory). Apparently, the instruction memory is physically separated from data memory. From a security perspective, ROM is much safer than RAM. Therefore, only one MPU is added on the right side to protect RAM. The main job of the MPU is to check whether there is an address out of range, misreading or miswriting. It was connected to the RAM through the logic gate. The kernel will send the address information to the MPU. Once the MPU checks that the address is out of range, the output will be blocked by the MPU and the data could be written to the RAM. In other words, the MPU sets the available address segment and generates a block. It then sends an interrupt signal if wrong address segment is used. Such approach is an external protection method to prohibit cross-border address data from being written to RAM so as to maintain system stability.

In addition, the internal design of the unit is roughly composed of registers. For memory chips, all memory accesses are of equal priority (Silberschatz et al., 1988). The memory hardware does not know or concern which memory part is used. The design idea is to limit the base of the process. The process can only access memory locations that belong to it.

As shown in figure 16, it can be set by adding two registers to determine the range that the process can access. One is the base register and the other is the limit register. To specify, base register holds the smallest physical boundary memory address while limit register limits the range of addresses. Each memory access of a process will compare the address range defined by the two registers. If the process visits wrong address range, it should be blocked as a way to avoid errors (Silberschatz et al., 2009).
Figure 17. Logical judgment block diagram (Silberschatz et al., 1988)

Figure 17 presents a more detailed hardware design, in which the address is checked by two logical judgments. Only when both logical decisions are YES can the access be allowed into memory. For logical decisions in NO, the MPU will block the access and trigger to interruption. After that, the blocked task will be handled by interruption service routine of the system. Moreover, operating system can access all the memory, where switching code and data in/out of memory is of great significance. Apart from that, the operating system kernel only allows changing the contents of the Base address and limit registers. To further improve the memory protection, the system could simplify MPU structure and add MPU without block functions to protect ROM. When an out-of-range command is detected, there will be a signal like interrupt signal indicating that MPU has figured out the command to be invalid. As for the internal structure of the MPU, the block diagram visually displays the components of MPU.

Figure 18. MPU block diagram

As illustrated, internal components of MPU are registers. The task_id register shows the running task, so that MPU knows which task is running. A total of eight sets of registers could set address segments for eight tasks at the same time. For example, when task_id equals 0, task0 is running. The base0, Limit0 and authority0 will be set for task0 (For the convenience of presentation, three registers are grouped together and named by group0). At the same time, all other sets of register for task0 will be closed, indicating that task0 can only access the same address with group0. If task0 wants to access another group, the MPU will block such attempt. Moreover, the choice register is used to operate more address with less register. The En_MPU register applied to open or close MPU. When En_MPU equals 1, MPU is open. When En_MPU equals 0, MPU is closed.
Above figure shows hardware MPU code in VHDL. All VHDL codes are written with ISE. Besides, it presents ports of MPU, which are used to connect with CPU. To be more specific, following table lists the function of ports.

<table>
<thead>
<tr>
<th>Port</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>i_adr</td>
<td>Instruction address</td>
</tr>
<tr>
<td>Stb_rom_i</td>
<td>Command chip selection</td>
</tr>
<tr>
<td>Int_ins</td>
<td>When instruction range is out of range, it is generated high level</td>
</tr>
<tr>
<td>D_dat</td>
<td>Externally written data</td>
</tr>
<tr>
<td>D adr</td>
<td>External incoming address</td>
</tr>
<tr>
<td>D_rd_dat</td>
<td>Internal outgoing data (for reading)</td>
</tr>
<tr>
<td>Ram_addr_bus</td>
<td>Address of RAM</td>
</tr>
<tr>
<td>We_i</td>
<td>Write signal (write operation generate high)</td>
</tr>
<tr>
<td>Stb_i</td>
<td>Chip select signal</td>
</tr>
<tr>
<td>Stb_ram_i</td>
<td>Chip select signal to access RAM</td>
</tr>
<tr>
<td>Stb_real_ram_i</td>
<td>MPU Chip select signal (If it generates low level, MPU block access to RAM)</td>
</tr>
</tbody>
</table>

Through these interfaces, PicoBlaze's data bus and address bus are connected to the MPU to control the MPU work.

The figure above illustrates all registers set in MPU. The En_MPU signal is used to control the open/close of MPU like a switch. The selection signal is multiplex register, which can operate more address with less registers. Moreover, the signal rule is used to judge logical selection. If the rule equals 1, the access is proved to pass all logical checks of MPU. If the rule equals 0, MPU will block this access to RAM.

Figure 21 shows the example of task0. When both rule and stb_ram_i equal 1, the stb_real_ram_i will generate 1, indicating MPU allows the access to RAM. In other words, when there is memory access, the stb_ram_i will generate 1. The MPU will check address of this access. If the address is valid, the rule will equal 1 and stb_real_ram_i will generate high level by MPU. If invalid, the rule will equal 0 and stb_real_ram_i will generate low level by MPU, which means that MPU blocks this access.

After introducing different register functions, how processor controls MPU and writes data to MPU register will be explored then. As mentioned before, CPU is connected to MPU through parallel buses (address bus and data bus). According to core address decoder, only four different addresses from 11011000 to 11011011 (00 01 10 11) are available to control MPU. Different addresses are assigned to different registers, thus the address decoder decodes address information sent by CPU and generates chip select signal, which can be identified by MPU. In addition, MPU will load different address into different registers.
Since MPU has many registers, there remains a problem on how to use four addresses to control all registers. To solve that, a register namely choice is used to realize address multiplexing. As showing in the figure, the MPU uses two lowest addresses to distinguish four addresses transported via parallel buses. To specify, the first address 11011000 is used to set En_MPU register. The second address 11011001 is used to set task_id register. The third address 11011010 is used to set choice register. The fourth address 11011011 is used to set base and limit registers according to the value of choice register. For example, the choice (6) equals 1. When choice equals 0, the zero bit of data will be written into the eighth bit of ins_base0 register while the first bit will be written into ninth bit of ins_base0. The value of choice is [1][1][0][0][0][0][0]. When choice equals 1, the value of choice is [1][1][0][0][0][0][1], and all eight bits data will be written into remaining 8 bits of ins_base0 register.

Software Design

The software part of the project is to test basic functions of MPU. In that case, it will write code with OpenPICIDE, by which the software can convert assembler to VHDL. Moreover, it can load instructions that can be recognized by PicoBlaze into the ROM file, from which the processor can read instructions. Additionally, the communication protocol is to use assembly instruction set.

Generally, three main assembler instructions are used to load data into register. The first operand of load instruction must specify the register to be loaded as register ‘s’ followed by a hexadecimal digit. Next, the second operand specifies a second register value in a similar way. As Figure 23 shows, the value 0x10 is loaded into register S0 through load instruction. The out instruction enables the contents of any registers transferring to logic external of PicoBlaze. Moreover, the port address can be defined by a constant value. Each out instruction specifies the source register as ‘s’ followed by a hexadecimal digit. After that, it further specifies the output port address by using a register value in a similar way. Seen from the figure above, the value of register S0 0x10 is transferred to memory address 0x10. The In instruction enables data values external to PicoBlaze transferring into any one of internal registers. This instruction is contrary to the out command. The value 0x10 of memory address S1 is transferred into register S0 through In instruction. Thereby, the final value of register S0 is 0x10.

Test methods and code

Test 1
As Figure 24 shows, the first test aims to test the value that can be loaded into register and the read value from the register. In the test, each task is limited by one register. For example, task0 is limited by base0, limit0 and authority0. The right tests whether the value can be read from MPU to register two.

Test 2

Test 2 aims to test block function. At first, a register is used to show the value of the task, indicating that it enables MPU knowing which task is running. For example, a value of 0 means the task0 is running. The register base0, limit0, authority0 (Read/Write) should be opened for task0. At the same time, bases from base1 to base7, limits from limit1 to limit7, and authorities from authority1 to authority7 should all be closed. Task0 can only access base0 and limit0. If the address is valid, MPU will release strobe signal to allow data written into RAM. If task0 tries to access other base or limits registers, MPU will cut off the strobe signal to stop any writing operation. In this way, MPU provide memory protection for RAM in a way avoiding wrong data written into RAM.

The block function test is as shown in the left side of Figure 25, the base address is set as 0x000 while limit address is set as 0x040. The effective address range set for task 0 is from 0x000 to 0x03f. Besides, the authority indicates functions in reading and writing. While the right side is to set address for task1. The base1 sets address 0x040 and limit1 sets 0x040. Therefore, the effective address range set for task1 is from 0x040 to 0x07f. However, the authority only sets read function, which means reading operation is allowed within the valid address range. Writing operations are excluded. In addition, with regards to testing coverage, all addresses can be tested by looping. The order of testing starts with performing a full address read test on task 0, and followed by a full address write test of task 0. After that, it performs a full address read test on task 1, and continues with a full address write test of task 1.
The former is read test while the later is write test. Apparently, adding instruction can perform overlay calculations to accumulate all addresses.

Test 3

The third test is to interrupt test code. When MPU generates a block, it will immediately call this part of the interrupt instruction to issue an interrupt signal. From the aspect of improving the follow-up system, ISR (Interrupt Service Routine) spares certain time for the system to call instructions in order to handle blocked tasks. The right side shows the last test is for ROM protection. Task0 can execute instructions from 0x100 to 0x1ff and task1 can only execute instructions from 0x200 to 0x2ff. Once there are instructions indicating the address is out of range, signal int_ins will be high.

RESULT AND EVALUATION

The results are presented by waveform picture that is simulated in ISE. It enables monitoring relevant signals for different clock cycles to verify if the design has met requirements of the project.

Test 1 Simulation Results

The first test is concerning the basic function of register. It only proves that the register can be loaded and read. In other words, the value of register update along with the clock cycle.
Figures above pictorially show that the value updates, proving that register of MPU can load and read value normally.

**Test 2 Simulation Result**

For the convenience of monitoring, the results of Test 2 and Test 3 are combined to present in the same waveform.

The En_MPU signal is the enabler signal of MPU, working like a switch of the MPU. When the signal is high, MPU is turned on and starts to work. When the signal is low, MPU is off. Moreover, CPU generates the stb_ram_i signal, indicating that CPU tries to write data into RAM. The stb_real_ram_i signal is generated by MPU. When MPU checks the valid address assigned by the task access, the stb_real_ram_i signal will be either in high level or low level.

Earlier in the hardware design section, MPU uses logic AND gate to do logical judgment. Both stb_ram_i signal and stb_real_ram_i signal are generated in high level. The strobe signal would pass MPU while CPU could write data into RAM. Furthermore, the task_id signal shows running status of a task. If the task_id equals 0, task0 is running at present. Additionally, The we_i signal shows current type of operation. For example, high we_i signal indicates current operation is writing. When it is low, the current operation is reading.

The figure above shows that MPU is open to test task0. The valid address for task0 is from 0x000 to 0x03f. At the beginning, the address is 0x000 and task0 execute a reading operation in the effective address range, thus the we_i signal is low. In addition, both stb_ram_i signal and stb_real_ram_i signal are high, proving that MPU checks access and confirms its validity. The strobe signal has passed the MPU.

Each address is tested by looping. When the address is 0x040 and not valid for task0, the MPU generates low signal stb_real_ram_i. It proves that MPU has executed block function. After that, the int_mpu signal is generated to be high.
Moreover, the register SC is used to count the number of interrupts. Whenever an interrupt is triggered by a block, the value of SC will be incremented by one. As it is shown in the figure, after first interrupt signal, the value of SC is updated to 1.

![Figure 32. Waveform of task0 blocked](image)

Until the test reaches address 0x07f, the read test for task0 is complete. The valid address for task0 is from 0x000 to 0x03f. Moreover, the address from 0x040 to 0x07f is set for task1. In this way, all read operations executed between 0x040 and 0x07f are blocked by MPU to generated interrupt signals. Therefore, the read test for task0 is correct, which proves MPU working normally.

![Figure 33. Waveform of task0 write operation](image)

The above test is concerning the write operation for task0. Writing operation starts from the initial address at 0x000. The we_i signal generates high level from the beginning, indicating that it is executing writing operation. Until the address 0x03f is valid, MPU always allows the access to pass. Both stb_ram_i signal and stb_real_ram_i signal are high.

![Figure 34. Waveform of task0 blocked](image)

For task 0, starting from address 0x040 is invalid. So, MPU blocks all writing operation from 0x040 to 0x07f. The signal stb_real_ram_i is generated at a low level. After each block, MPU also triggers the interrupt. Moreover, task0 is to test reading and writing operations for all address. As far as the test results are concerned, MPU blocks the normal detection address and
triggers the interrupt. The following test is for task1. However, task1 is set only with read authority, which means that it can only execute reading operations within effective address and could execute any writing operations.

![Figure 35. Waveform of task1 blocked](image)

Seen from the test sequence, the primary test is still the read testing for task1. The valid address of task1 is from 0x040 to 0x07f. It can be acquired from the figure that both we_i signal and the stb_real_ram_i signals are low. The reading operation is blocked and triggered in interruptions by MPU. Additionally, the task_id equals 1, proving that task1 is currently running.

![Figure 36. Waveform of task1 blocked](image)

From 0x040, MPU does not block task 1 to execute reading operations. Both stb_ram_i signal and stb_real_ram_i signal are generated at a high level.

![Figure 37. Waveform of task1 authority test](image)
The above test is to test writing operation of task1. As mentioned, task1 only has read authority, thus it could not execute any writing operations. From 0x000, the \text{we}_i\ signal is generated at a high level, indicating that task1 executes writing operation. The \text{stb\_real\_ram\_i} is generated at a low level, indicating that MPU blocks access and triggers the interrupt.

![Figure 38. Waveform of task1 blocked](image)

Even it starts from the address 0x040, MPU still blocks the write operations of task1. MPU always triggers an interrupt signal after blocking.

**Test 3 Simulation Result**

![Figure 39. Waveform of whole block test](image)

Such figure provides an overview of whole block test. Test result has confirmed that MPU can implement memory protection function properly. When it detects an out-of-range address access, it could block the access and trigger an interrupt signal. From this respect, it basically achieves the expected function of memory protection. Following test is for ROM protection testing. As mentioned, ROM could not perform any writing operations. Instead, it only supports reading operation. So, MPU can be simplified when comparing to RAM. The blocking function is removed while the detection address function is retained. When MPU detects an out-of-range address, it will send a high-level signal \text{int\_ins}.

![Figure 40. Waveform of ROM test](image)

Task0 executes instruction at address 0x200. MPU generates \text{int\_ins} at a high level. The task1 execute instruction at address 0x104 and MPU generates \text{int\_ins} at a high level. Moreover, MPU of the ROM does not link with interrupt. The reason is that the interrupt signal actually lags behind the clock cycle of the instruction access. That is to say, when MPU gives out interrupt signal, the instruction that crosses the boundary has already completed the read operation in the ROM. In this case, interrupts would waste the clock cycle of the system. In addition, the interrupt signal is to call the interrupt service routine at the system level.
CONCLUSION

In general, the proposed embedded strategy works out successfully. The hardware MPU is designed to protect memory. As a fact, original system design did not involve a complete memory protection mechanism. Misreading and miswriting of code could easily lead to address crossing. Once address crossing occurs, there will be inevitable unknown errors to the system. To make a complete and safe system, complete memory protection mechanism is desired. The test result has succeeded in implementing the expected memory protecting function. This design meets both operational capabilities of CPU and required functions at the lowest cost.

The key point of this design is in hardware. The MPU checks the address of the running task while supporting the multitasking operation. Once the invalid address is figured out, the access signal is immediately blocked to prevent the erroneous data from writing into the memory.

This research is based on RTOS designed by Adam (Matthews, 2018). The aim of this research is to make such OS better by adding new functions. More importantly, this research fills the gap since there is currently no complete memory protection available for operating systems based on PicoBlaze soft-controller. With respects to design requirements, since the system is a small-embedded operating system and many protection schemes are not capable to be applied in the system, current design is considered as the best solution for such system in all aspects. In addition, prior to the research, there has been a lack of software system with memory protection function designed based on PicoBlaze. Therefore, the research made certain contributions to the expansion of the PicoBlaze software system. However, the researcher failed to achieve full OS functions. In that sense, OS functions could a researching field in future work.

Admittedly, the research only innovated in the hardware aspect. However, standing from entire software system, the research has not been completed. Full OS functionality needs to be supplemented in the software section. Moreover, the most important field is how to add MPU to the system. After creating tasks, the system assigns them different private address segments. MPU will accept these settings and start working. Every task that runs from the ready list and goes to the run list must be checked by MPU. Once an out-of-range address is found, the access is blocked and the interrupt signal is triggered and processed by ISR (Interrupt Service Process). In addition, designing the waiting list is also necessary. Tasks that have finished running or been interrupted would be placed in this area waiting for the system to process.

Another area that requires to be extended is to add some memory protection features when communicating between tasks. For example, when communicating, the system must allocate them a shared memory. However, the problem is that two tasks could not access the shared memory area at the same time, which might cause system error. Therefore, a protection mechanism is expected to compensate the limitation. When a task is using shared memory, the second task is not allowed to access until the current task finishes running and leaves shared memory area. Finally, future researchers can choose other types of soft-controllers to construct more complex systems.

REFERENCE


Freeman, M. (2005). Designing a ubiquitous computing development kit (pp. 97-105). IET.


The 19th International Conference on Electronic Business, Newcastle upon Tyne, UK, December 8-12, 2019