XAPP738 (v1.0) February 22, 2008

Code Acceleration with an APU Coprocessor: a Case Study of an LPM Algorithm

Contact: Glenn Steiner

Summary

In network address routing, an IP packet is routed to a specific destination based on its IP address. Frequently, a variant of the longest prefix match (LPM) algorithm is implemented. This algorithm looks up IP addresses in a routing table and finds a forwarding path for the incoming IP address based on the longest possible match from the routing table.

This application note describes both hardware and software implementations of a simple LPM algorithm, consisting of a linear search of the routing table for the best match. The linear search sequentially compares each entry in the routing table to the IP address being looked up and remembers the longest matching prefix encountered. While more sophisticated algorithms can be implemented, the linear search provides a valid comparison between the various methods of storing and accessing the routing table. The storage and retrieval of the routing table data is implemented in software and hardware in four different ways:

- Processor Local Bus (PLB) attached Double Data Rate (DDR) memory
- PLB-attached block RAM
- On-chip memory (OCM) attached block RAM
- Auxiliary Processor Unit (APU) attached block RAM

This application note compares the speeds of these software implementations. In addition, to accelerate the lookup, a coprocessor is implemented in the Virtex™-4 FPGA to carry out each lookup as an APU user-defined instruction. This hardware implementation achieves a substantial improvement over the software implementations.

Introduction

An LPM algorithm is an essential part of a router. This application note describes hardware and software LPM implementations that use the APU interface. It also describes how the APU interface can be used as an alternative to DDR and OCM memories. This application note compares the performance of LPM lookups performed on both DDR- and APU-attached memories as well as on a custom APU-attached coprocessor used to accelerate lookups. This application note shows the benefits of using the APU interface to communicate with memory, especially when used in conjunction with application-specific hardware. The LPM algorithm provides a typical application of what users need in their own designs.

This application note defines the LPM algorithm and the APU to Block RAM Fabric Coprocessor Module (FCM). It describes the implemented reference design and provides its performance results.

Key Concepts

Longest Prefix Match Algorithm

The longest prefix match algorithm comprises a number of algorithms used to find the closest match of an IP address in a routing table. The input IP address that is being compared to the entries in the table is known as the *search key*. The 32-bit IP address specifies the destination of an IP packet in internet protocol routing. There can be more than one way for an IP packet to reach its final destination in a system. Thus, routers attempt to select the best intermediate destination (next hop) from any number of possible next hops. The LPM algorithm determines this next hop by performing a lookup in a routing table.

In the linear search version of the LPM algorithm, every entry of the table is examined in sequential order. The only input is the search key, which is compared against each entry of the table. An entry becomes the new longest match if it matches the search key and has a prefix that is at least as long as any previous matches. When every entry in the table has been searched, the index pointing to the IP address with the longest matching prefix is known and can be used to fetch the entry, when needed. This index to the longest matching prefix entry is the output of the search.

Routing Table

Figure 1 shows the format of an entry in the routing table. The text in parentheses indicates the code name of each 32-bit field.

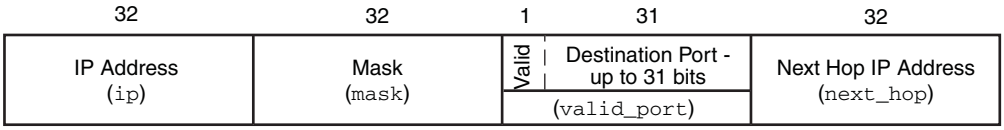


Figure 1: **Routing Table Entry Format**

Each entry in the routing table has the same format. The first word of the entry is the IP address (`ip`). The search key (`ip_to_find`), which consists of a 32-bit IP address, is compared against this first word of the entry.

The mask field (`mask`) is a 32-bit binary string starting with `1`s at the most-significant bits and terminating in `0`s, if any. Bits that are set to `1` are used in the comparison. For example, if a mask is composed of 28 `1`s followed by 4 `0`s, the 28 most-significant bits of the IP address and the search key are used in the comparison. The mask can only contain sequential `1`s followed by `0`s, if any. The most-significant bits are `1`s until the first `0`, at which point the remaining bits must also be `0`s. In this application note, these values have been chosen as the mask:

- `0xFFFFFFFF` (all bits of the IP address word and the search key must match)
- `0xFFFFFFF0` (the most-significant 28 bits of the IP address word and the search key must match)
- `0xFFFFFF00` (the most-significant 24 bits of the IP address word and the search key must match)
- `0xFFFFF000` (the most-significant 20 bits of the IP address word and the search key must match)
- `0xFFFF0000` (the most-significant 16 bits of the IP address word and the search key must match)

The mask ultimately determines the longest prefix. Even if an entry has an exact match to the search key, the length of the entry's mask determines how many bits are compared. The longest possible match is a 32-bit match with a mask of `0xFFFFFFFF`.

The third word in the routing table entry contains the Valid bit and the destination port number (`valid_port`). The Valid bit must be set for the entry to be considered valid and included in a lookup. The remaining 31 bits of this field specify a destination port for the matching IP address of the entry. In typical routers, much fewer than 31 bits are needed for this field because the number of physical ports on a router is relatively small. This reference design uses 31 bits to enable word alignment.

The fourth word in the routing table entry is the next hop IP address (`next_hop`). It is not used in the comparison but is used in forwarding the packet. This field is not read during the comparison operation because it is not needed until after the comparison is completed.

The routing table in this design uses 64 KB of space, which is nearly the entire 72 KB total space available on the XC4VFX12 device. The table was designed to be as large as possible to illustrate the different methods to store and access data. Each entry is 4 words (or 16 bytes), which results in 64 KB/16 bytes, or 4K entries. Entry 0 is used as a *no match found* location, so the total number of entries in the table is 4095.

The 16 most-significant bits of every entry's IP address are set to `1`, followed by a 16-bit pseudorandom number. A typical entry has an IP address such as `0xFFFF7AC1`. The pseudorandom number has a value between `0x1` and `0xFFFF`. It is generated using a software implementation of a Linear Feedback Shift Register (LFSR). Therefore, each of the 4095 entries has a unique IP address. This code segment shows the generation of the pseudorandom number:

```
Xuint32 LPM_random(Xuint32 number){
  Xuint32 tap_1=0;
  Xuint32 tap_2=0;
  Xuint32 tap_3=0;
  Xuint32 tap_4=0;
  if(number & 0x8000)
    tap_1 = 1;
  if(number & 0x4000)
    tap_2 = 1;
  if(number & 0x1000)
    tap_3 = 1;
  if(number & 0x0008)
    tap_4 = 1;
//the next random number in the sequence is generated by shifting
//the number left 1 bit XORing the taps to achieve the LSB.
return ((((tap_1 ^ tap_2) ^ tap_3) ^ tap_4) | (number << 1)) &
0x0000FFFF;
}
```

Each entry's mask has four fewer `1`s set than in the previous entry's mask. The shortest comparison mask implemented is `0xFFFF0000`, after which it starts over from `0xFFFFFFFF`.

The Valid bit is set in every entry added to the table. The destination port and the next hop IP address are not used in the lookup comparison. For convenience, they are set to be the current entry number. The ninth entry in the table has a destination port of `0x00000009` (31 bits) and a next hop IP address of `0x00000009` (32 bits).

LPM Search

Figure 2 shows how the comparison is performed in the linear search LPM.

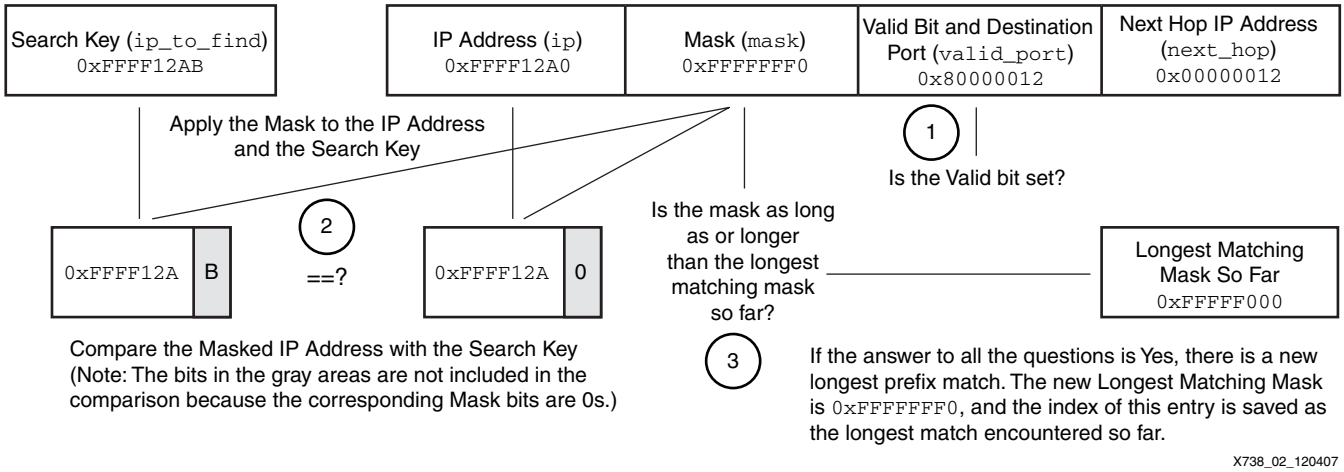


Figure 2: **LPM Comparison**

For an entry to be considered a match to the search key, it must satisfy these three conditions (refer also to the numbers in Figure 2):

1. The entry's Valid bit must be set.
2. The entry's masked IP address and the masked search key are equal to each other.

   An IP address is masked by performing a bitwise AND of the mask and the IP address.
3. The entry's mask is greater than or equal to the mask of any previously matched entry.

When all three constraints are met, the entry becomes the longest prefix matched so far. The index of the entry in the table is stored along with the mask for use in the comparisons to entries later in the table. This LPM search algorithm can be performed in hardware and software. This code snippet is the core of the LPM lookup:

```
//search the entire table, indices are separated by 4 (4 words per entry)
for(i=4; i<SEARCH_MAX_APU; i = i+4){
  //read an entry from APU attached Block RAM
  ip = read_bram(i);
  mask = read_bram_next();
  valid_port = read_bram_next();
  //no need to read next hop here.
  //Key LPM comparison
  if(((ip_to_find & mask) == (ip & mask)) && (mask >= curr_mask) &&
      (valid_port & 0x80000000)){
    //we have a valid match
    curr_mask = mask;
    matched_i = i;
  }
}
```

When the APU-attached block RAM is used, the entries are read by making calls to `read_bram(address)` and `read_bram_next()`, as discussed in "APU to Block RAM Controller." The comparison for the three conditions is performed in this line:

```
if(((ip_to_find & mask) == (ip & mask)) && (mask >= curr_mask) &&
(valid_port & 0x80000000)){
```

Table 1 breaks down the code within the line.

Table 1: **Code Breakdown**

Code	Description
`ip_to_find & mask`	Masks the search key
`ip & mask`	Masks the IP address of the current entry
`(ip_to_find & mask) == (ip & mask)`	Ensures that the masked IP addresses match
`mask >= curr_mask`	The mask of this entry is at least as long as the mask of the previous matching entry
`valid_port & 0x80000000`	Masks out the most-significant bit, which is the Valid bit

When these operands are combined using a logical AND, each operand must be True for the conditions constituting a match to be met.

Other LPM Algorithms

The linear search is neither the best nor the only LPM algorithm available. The linear search algorithm was chosen to demonstrate the disparity between the different memory accessing schemes. The linear search requires many memory accesses for a single search. Three out of four words in every entry must be read. Thus, the differences between two speeds of memory accesses are easily demonstrated. Implementing a large data table is necessary because the PowerPC® processor contains a 16 KB data cache. When the entire table can fit in the cache,

performance results for DDR-based searches are skewed and not representative of a larger data set. Also, routing tables can have hundreds of thousands of entries, much more than can fit in this reference design's block RAM.

Another well-known LPM algorithm, known as a trie (prefix tree) implementation, was also examined in software and hardware. This algorithm's lookup time depends on the size of a single entry rather than the size of the entire table (as the linear search does). This algorithm is very efficient in terms of speed, although a large memory overhead is required to store the tree structure used in searches. When this algorithm is implemented, due to the relatively few memory accesses and slightly higher overhead from surrounding code, the performance differences between utilizing different memory-access types are not clear. For this reason, the trie implementation was not considered for this application note.

APU to Block RAM Controller

Overview

The APU to Block RAM controller is a custom FCM that attaches to the APU interface and to 64 KB of block RAM. Figure 3 shows a simplified block diagram of this controller.

Figure 3: **Simplified Block Diagram of the APU to Block RAM Controller**

The module attaches to the APU interface over a Fabric Coprocessor Bus (FCB) and uses the signals described in UG018, *PowerPC 405 Processor Block Reference Guide* to communicate with the rest of the system. Block RAM, which is instantiated within the FCM, is controlled by a state machine. All input and output accesses from the module use the FCB.

The APU to Block RAM controller has two distinct functions. First it operates as a memory controller that allows APU-issued instructions to read and write the block RAM contained in the module. Using APU user-defined instructions (UDIs), commands can be issued that write to a specific address in block RAM, read from a specific address in block RAM, and read from the next sequential address in block RAM. These three instructions emulate the function of a memory controller. They are implemented as a sequential state machine design that toggles the appropriate signals to the APU, forwards operands to the block RAM, and decides when to assert the write-enable signal.

The second function is to perform a hardware-based LPM lookup of an IP address in the block RAM. By issuing a single UDI and supplying a search key, an LPM lookup can be performed as a single hardware instruction. This is much more efficient than issuing independent reads and writes to the block RAM that are required in software implementations of the algorithm. The hardware LPM is incorporated into the same state machine as the other function of the controller, the two functions do not share any states except for the IDLE state.

APU User-Defined Instructions

The APU coprocessor can decode up to eight UDIs. The reference design implements four UDIs. The first step in implementing a UDI is to set the initial value for the APU controller configuration register. In this design, the value is set to `0x1E01`, using the Embedded Development Kit (EDK) tool suite, where:

- FPU instruction decoding, FPU complex arithmetic instruction decoding, FPU conversion instructions, and FPU estimation instructions are disabled because the FPU is not used in the reference design.
- FCM usage is enabled.

Refer to UG018, *PowerPC 405 Processor Block Reference Guide* for detailed information on the APU controller configuration register.

Next, the individual UDI configuration registers are programmed as shown in Table 2. RaEn and RbEn enable the use of general-purpose registers *a* and *b*. These registers are the second and third arguments given in the assembly instructions. They contain any relevant operands for the instruction. GPRWrite signifies that this instruction writes a result to a general-purpose register. It is the first argument of an assembly instruction and is the target register.

Table 2: **UDI Configuration Registers**

UDI	Name	Configuration Register	RaEn	RbEn	GPRWrite	Type	UDIEn
0	write_bram	0xC07605	X	X		Autonomous (10)	X
1	read_bram	0xC47501	X		X	Blocking (00)	X
2	read_bram_next	0xC87101			X	Blocking (00)	X
3	lookup_apu_table	0xCC7501	X		X	Blocking (00)	X

The first 12 bits determine the instruction's opcode. The next three bits [12:14] determine which registers are needed in performing the instruction. Bits [26:27] determine the type of instruction of the UDI, and the last bit enables the instruction for use. Table 2 shows the various characteristics of the four instructions. Their names indicate their functions. The write_bram instruction is the only autonomous instruction of the four. That is, after the APU issues this instruction, the PowerPC processor can resume work without waiting for the instruction to finish. The remaining three instructions require the CPU to be stalled because they need to return the result of a lookup or read.

A set of predefined instructions to be used as UDIs is located at `project/ppc405_0/include/xpseudo_asm_gcc.h`. This code is a sample definition from the file:

```
#define UDI0FCM_IMM_GPR_GPR(a, b, c)                    \
        __asm__ __volatile__("udi0fcm " #a ",%0,%1" : : "r"(b), "r"(c))
```

Similar definitions exist for all the combinations of IMM and GPR for each of the eight UDIs in this file. An IMM field contains an immediate (5-bit number) value, and a GPR is a general-purpose register that contains a 32-bit value. In the instruction shown, the arguments *a*, *b*, and *c* translate to the target register and the two operand registers. Each can be replaced by an immediate value.

To create C-style functions that perform the appropriate UDI, wrappers for these pseudo-assembly instructions were created. The code for the four UDI wrappers is shown here:

```
//UDI 0, write 32 bits to the look-up table
inline void write_bram(unsigned int address, unsigned int data){
  UDI0FCM_IMM_GPR_GPR(0, address, data);
}
//UDI 1, read 32 bits from the look-up table
inline unsigned int read_bram(unsigned int address){
  unsigned int temp;
  UDI1FCM_GPR_GPR_IMM(temp, address, 0);
  return temp;
}
//UDI 2, read the sequentially next 32 bits from the look-up table (less
//clock cycles)
inline unsigned int read_bram_next(){
  unsigned int temp;
  UDI2FCM_GPR_IMM_IMM(temp, 0, 0);
  return temp;
}
//UDI 3 do a complete hardware lookup of the ip address. Returns an index of
//longest match.
inline unsigned int lookup_apu_table(unsigned int ip){
```

```
    unsigned int temp;
    UDI3FCM_GPR_GPR_IMM(temp, ip, 0);
    return temp;
}
```

These wrappers are created as inline functions, eliminating unnecessary function calls. Because the result of the pseudo-assembly function is stored in one of the arguments, the *temp* variable is used as an intermediate before returning the result in a more conventional manner. Table 3 describes each of the functions.

Table 3: **Wrapper Function Descriptions**

Wrapper	Description
write_bram	Writes the 32-bit data value to the specified address. The least-significant 15 bits of the address integer are used to select the location in the block RAM where the data is stored. Nothing is returned.
read_bram	Reads the 32-bit data value from the specified address. The least-significant 15 bits of the address integer are used to select the location in block RAM from where the data is read. The result of the read is returned.
read_bram_next	Reads the 32 bits of data at the address sequentially following the previously issued address (typically by a read_bram instruction or a previous call to the read_bram_next instruction). This instruction is used when general-purpose registers do not need to wait for operands (such as the address), enabling APU instructions to be executed faster. This instruction is useful because reading words from several addresses sequentially is a common operation and routing table entries are four words long.
lookup_apu_table	Performs a complete LPM search in hardware for the IP address specified in the search key. This function returns either the index of the longest prefix match or `0` if no match is found. Only this instruction is needed when using the hardware-based coprocessor to perform LPM lookups. However, the write_bram instruction must first be issued to fill the routing table, and then a read_bram or read_bram_next instruction must extract the individual fields of the matching entry. This instruction is implemented as a blocking instruction, which means that the PowerPC processor is stalled until the FCM cycles through approximately 4,096 cycles to complete this instruction. Alternatively, this instruction can be split into two separate UDIs, where the first autonomous instruction starts the lookup, and the second blocking instruction fetches the result of a finished lookup. Although this configuration requires one more UDI, it frees up the PowerPC processor while lookups are performed in the FCM.

apu_to_bram Logic Module

The `apu_to_bram` logic module controls the block RAM by managing read/write instructions issued by the APU. The state machine that manages these instructions first awaits a valid read or write instruction. When a read or write instruction is received, the state machine must wait until all needed operands (address or data) are ready. When these operands are ready (APUFCMOPERANDVALID is asserted) and the APU is free to proceed, the state machine goes to the READ or the WRITE state and subsequently returns to the IDLE state. Table 4 defines the signals that communicate with the APU.

Table 4: **apu_to_bram Logic Module Signals**

Signal	Direction	Description
APUFCMDECUDI[0:2]	Input	These signals indicate the starting point of any instruction. When the instruction is valid and the decoded instruction is a valid UDI, the decoded UDI can be examined to determine which of the four UDIs is going to be executed. APUFCMDECUDI is often only valid for a single clock cycle. Thus, it is also latched whenever valid so that the current instruction can be known in later cycles.
APUFCMDECODED	Input	
APUFCMDECUDIVALID	Input	
APUFCMINSTRVALID	Input	

Table 4: **apu_to_bram Logic Module Signals *(Cont'd)***

Signal	Direction	Description
APUFCMOPERANDVALID	Input	Typically, these signals are required before an instruction can complete execution. After receiving a valid UDI, the UDI cannot be executed until any needed operands are valid and the APU indicates that it is ready with the APUFCMWRITEBACKOK signal. Because the read_bram_next instruction has no operands, the APUFCMOPERANDVALID signal is not asserted before the read is performed. Thus the read_bram_next instruction is faster than the read_bram instruction. All instructions require the APUFCMWRITEBACKOK signal. Sometimes, these signals can arrive in the same cycle as the APUFCMINSTRVALID signals, in which case the state machine can proceed directly to the READ or the WRITE state.
APUFCMWRITEBACKOK	Input	
APUFCMFLUSH	Input	This signal is used to flush out any instructions that are in progress. Until an executing instruction receives the APUFCMWRITEBACKOK signal, receiving a flush must reset the state machine to the IDLE state.
APUFCMRADATA[0:31]	Input	These buses reflect the data contained in the three registers of any calling UDI. The data buses for general-purpose registers a and b are inputs to the instruction, while the 32-bit result is the output from the instruction. Not every instruction uses all three of these signals.
APUFCMRBDATA[0:31]	Input	
FCMAPURESULT[0:31]	Output	
FCMAPURESULTVALID	Output	The FCMAPUDONE signal must be asserted during the last cycle of any UDI. If the UDI returns a result, the FCMAPURESULTVALID signal is asserted during this same cycle. After these signals are asserted, there is always a dead cycle in the APU before another instruction is issued.
FCMAPUDONE	Output	
FCMAPUSLEEPNOTREADY	Output	This signal is asserted during execution of any UDI and only remains deasserted while the state machine is in the IDLE state. This signal forces the CPU to not sleep.

The signals in Table 4 as well as many signals internal to the `apu_to_bram` logic module control the operation of the state machine. Table 5 defines the states of the state machine.

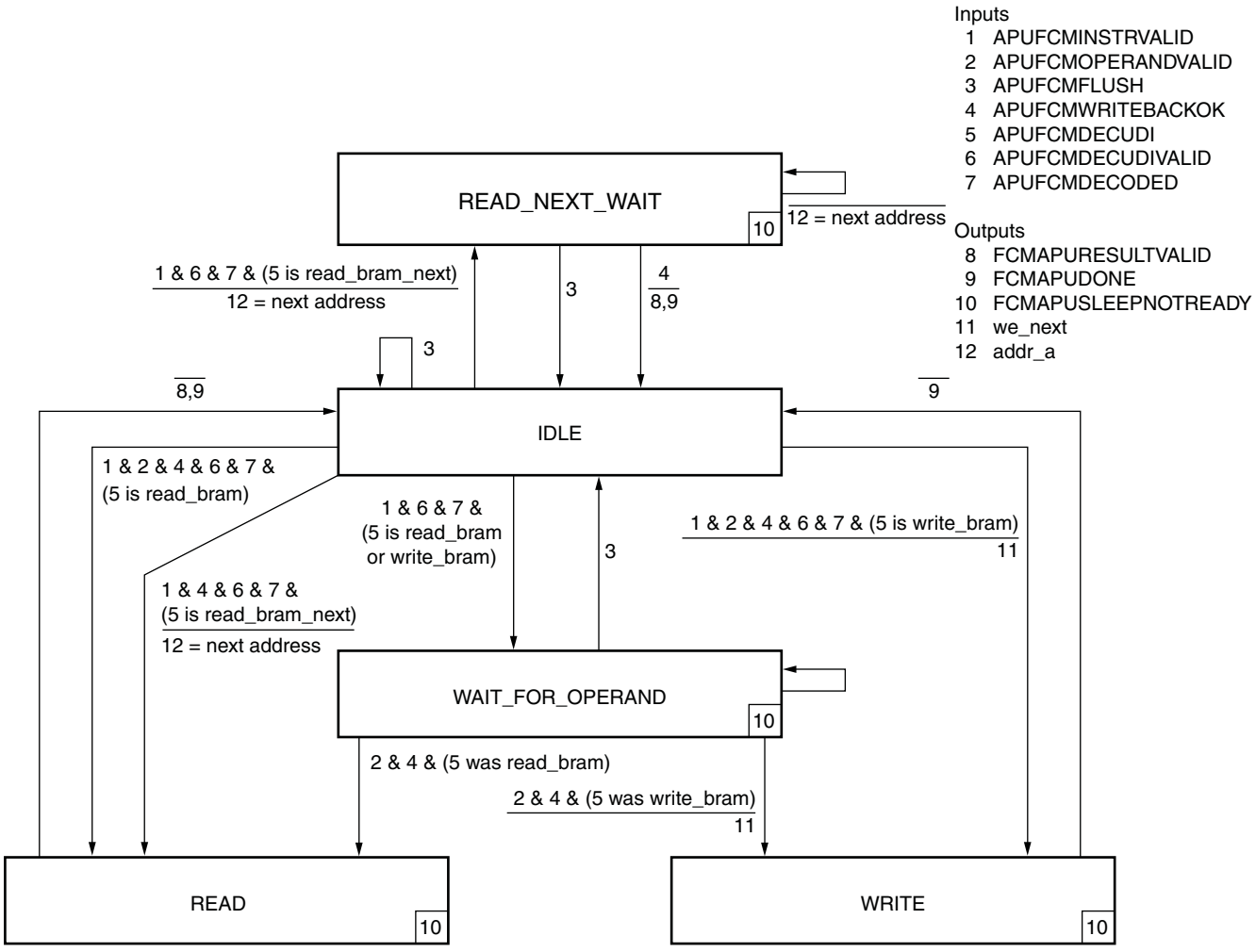
Table 5: **State Machine State Descriptions**

State	Description
IDLE	From this state, when a valid read_bram or write_bram UDI is received, the state machine proceeds to the WAIT_FOR_OPERAND state unless the operands are already valid, in which case the state machine proceeds to the READ or the WRITE state. If the instruction is instead a read_bram_next UDI, the state machine sets the block RAM address to be the next sequential address of the last address used, and either proceeds to the READ state (if the APUFCMWRITEBACKOK signal is asserted) or proceeds to the READ_NEXT_WAIT state (if the APUFCMWRITEBACKOK signal is not asserted). If a flush is issued during the IDLE state, the state machine ignores other signals and remains in the IDLE state.
WAIT_FOR_OPERAND	From this state, when the operands become valid and APUFCMWRITEBACKOK is asserted, the state machine transitions to the READ or the WRITE state as dictated by the original instruction. If a flush is issued during the WAIT_FOR_OPERAND state, the state machine ignores other signals and returns to the IDLE state.
READ_NEXT_WAIT	The state machine enters this state after a read_bram_next UDI is issued. The APUFCMWRITEBACKOK signal is not available during the same cycle. During this state, the block RAM address is set to be the next sequential address of the last address used. After receiving the APUFCMWRITEBACKOK signal, the instruction is executed, the result is validated, and the state machine returns to the IDLE state during the next cycle. If a flush is issued during this state, the state machine ignores other signals and returns to the IDLE state.
WRITE	Before this state is entered, the internal signal we_next is asserted, indicating that during the next cycle, while the state machine is in the WRITE state, the appropriate write-enable signal is also asserted. This is the final state during a write_bram UDI. Thus, the FCMAPUDONE signal is asserted, and the state machine returns to the IDLE state.
READ	This is the final state during a read_bram (and sometimes of a read_bram_next). The FCMAPUDONE signal and the FCMAPURESULTVALID signal are asserted, and then the state machine returns to the IDLE state.

Table 5: **State Machine State Descriptions *(Cont'd)***

State	Description
LOOK_UP	A lookup starts in this state. The address sent to the block RAM is set to be the index into the table. This index starts at `0x4` and increments by `0x4` each cycle up to `0x4004`. When the operands are valid and APUFCMWRITEBACKOK is asserted, the state machine transitions to the LOOK_UP_2 state.
LOOK_UP_2	The key LPM comparison is performed in this state. When the index is `0x4004`, the search has encompassed all 4095 entries of the table. The state machine indicates that the result is valid, and the instruction is executed. The state machine then returns to the IDLE state.

Between instructions, the state machine always returns to the IDLE state. Figure 4 shows the non-lookup states.



Transitions show asserted signals as the source of the transition. The transition with the most asserted signals has priority. A flush takes priority over all other signals.

X738_04_102307

Figure 4: **Partial State Diagram Showing Read/Write States**

LPM Hardware Accelerator

The second function of the `apu_to_bram` logic module is to perform a complete hardware-based LPM lookup. This hardware-based design has significantly higher performance than the software-based lookup using the same algorithm for these reasons:

- There are a number of processor instructions (incrementing pointers, checking loop conditions, doing a series of instructions per key comparison) that must be performed between reads of the block RAM in a software lookup. In hardware, these functions are done with logic inside the module itself.
- Each instruction issued to the APU coprocessor requires at least one dead cycle of overhead between APU instructions. Thus, a full table lookup requiring 12,000 read operations generates 12,000 dead cycles.
- Because the bus and general-purpose registers are 32 bits wide, software reads must be done in 32-bit units. However, for an LPM lookup, the hardware reads an entire entry (128 bits) during a single cycle.

The block RAM must be able to read an entire entry in a single cycle. Thus, the block RAM is arranged into four sections, the output of each representing a different 32-bit field of an entry. Software is required to access any of these four sections for reading or writing. Software uses the least-significant two address bits to determine which of the four sections to read or write a 32-bit value. During a hardware lookup, these two least-significant bits are not used. Figure 5 shows a high-level block diagram of this subsystem. The Data In and ADDR ports of all four block RAM sections have the same inputs. A write to a block RAM section occurs when its write-enable signal is asserted, and the least-significant address bits determine which location to write. Similarly, the least-significant address bits determine which block RAM's read data to return. If the instruction is an LPM lookup and not a read, the index is returned instead.

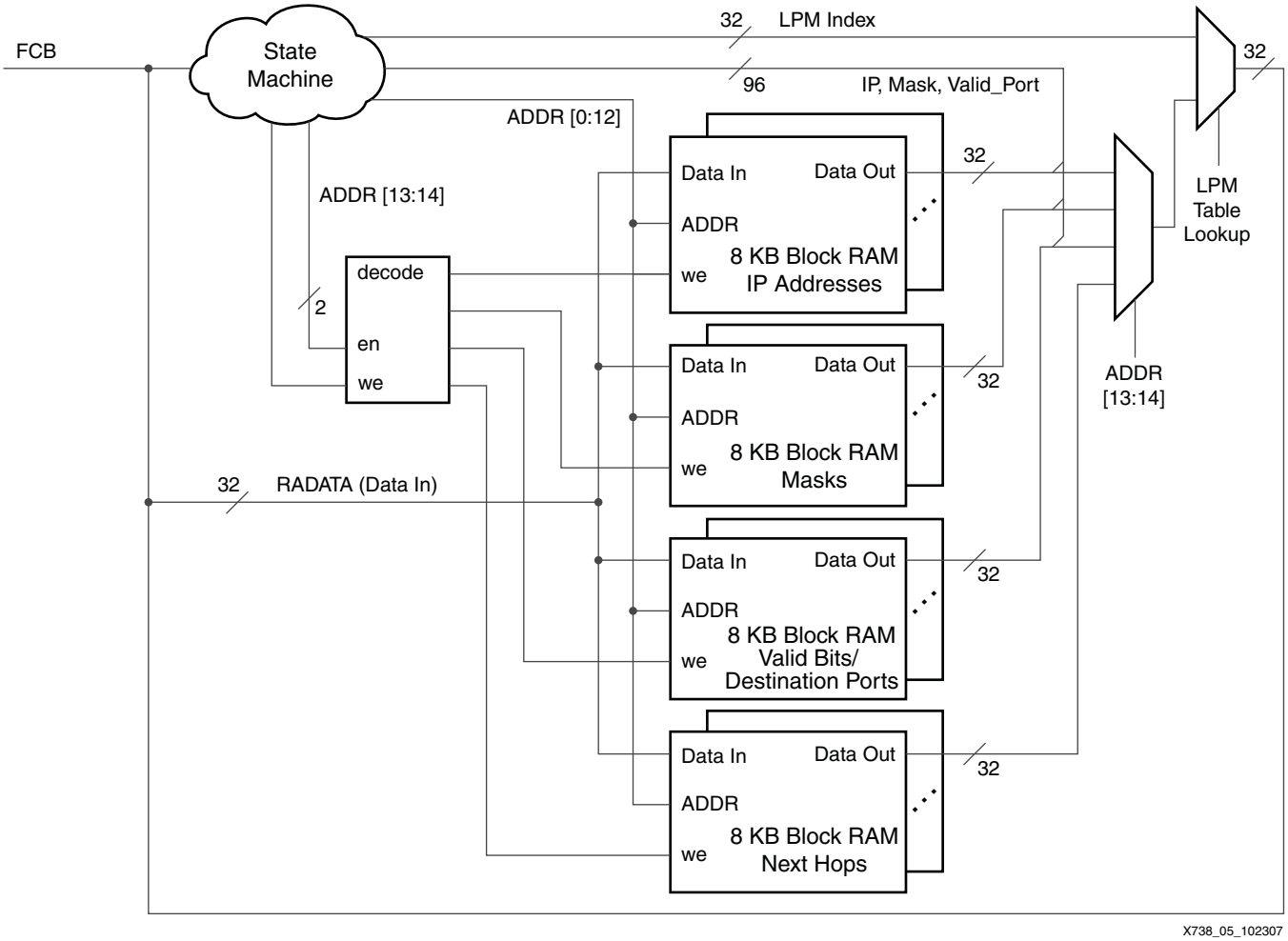
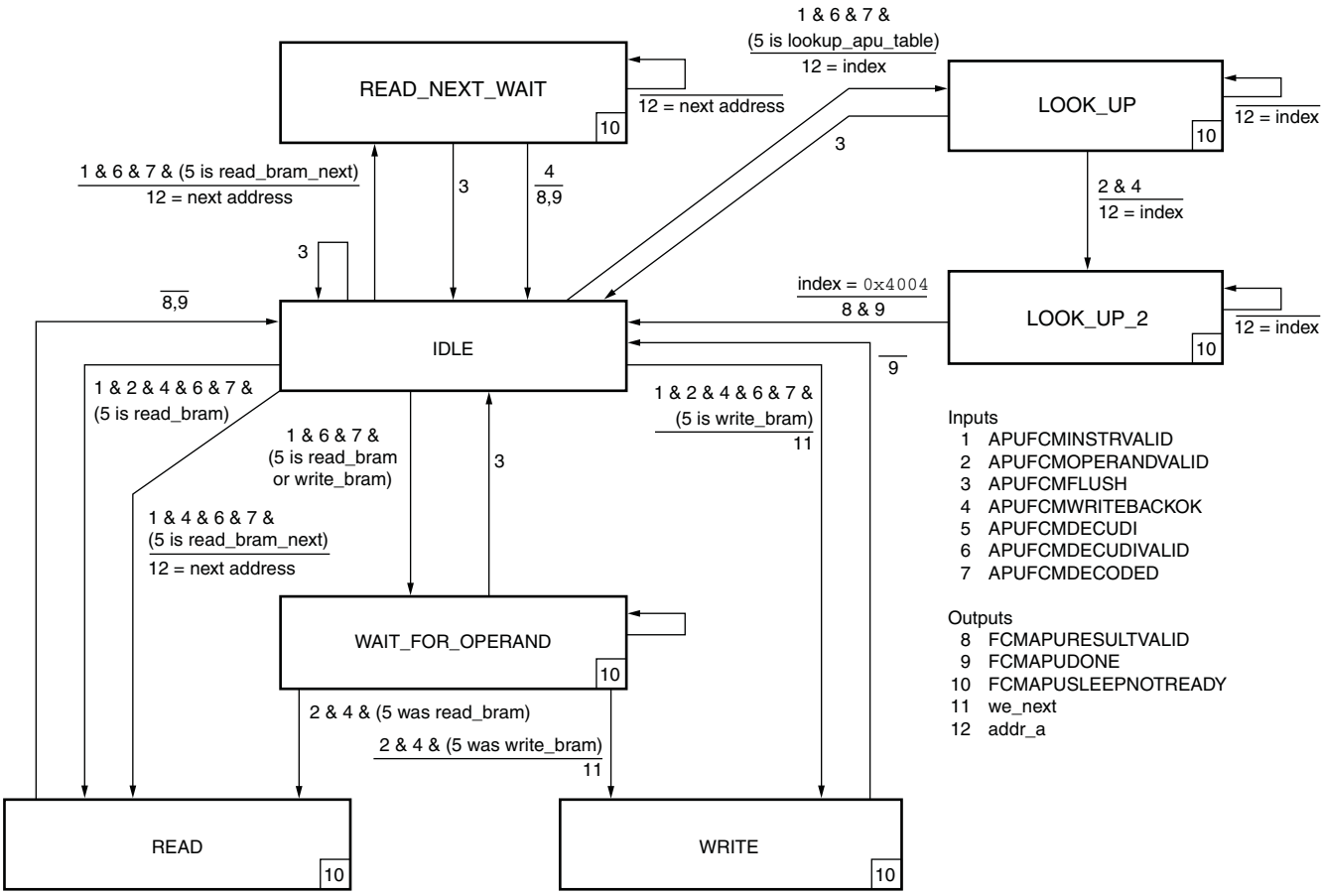


Figure 5: **Block Diagram of apu_to_bram Logic Module**

The LPM hardware lookup is part of the same state machine as the memory controller. While in the IDLE state, if a valid lookup_apu_table instruction is decoded, the state machine transitions to the LOOK_UP state. In the LOOK_UP state, the address sent to the block RAM is set as the index into the table. This index starts at `0x4` and increments by `0x4` each cycle up to `0x4004`. As soon as the operands are valid and the APUFCMWRITEBACKOK signal is asserted, the state machine transitions into the LOOK_UP_2 state. In this state, the key LPM comparison is performed. When the index reaches `0x4004`, the search has traversed all 4,095 entries of the table, the state machine signifies that the result is valid, and the instruction is executed before returning to the IDLE state. Because the instruction compares one entry per cycle and has approximately two to three cycles of overhead, the entire LPM search can be expected to complete in approximately 4,096 cycles, which translates to a predicted 41 µs per lookup, when the FCM frequency is 100 MHz. Figure 6 shows the state machine diagram.



Transitions show asserted signals as the source of the transition. When a transition is unclear, the transition with the most asserted signals has priority. A flush takes priority over all other signals.

X738_06_102307

Figure 6: **Complete State Diagram**

The code of the two lookup states is shown here:

```
//Start a lookup, wait for operands to be valid
LOOK_UP:begin//16
  ADDRA=index;
  FCMAPUSLEEPNOTREADY =1;
  //A flush sends returns to the IDLE state. Writeback ok not yet received.
  if(APUFCMFLUSH) begin
    next_state = IDLE;
  end
  //look up can now begin
  else if(APUFCMWRITEBACKOK & APUFCMOPERANDVALID) begin
```

```
    next_state = LOOK_UP_2;
  end
end

//longest prefix match is now underway, search entire table
LOOK_UP_2:begin                    //32
  ADDRA=index;
  FCMAPUSLEEPNOTREADY =1;
//masked ip_to_find matches masked ip. mask is at least as long as the
//longest matching mask. Valid bit is set
  if(((ip_to_find & mask) == (ip & mask)) && (mask >= curr_mask) &&
(valid_port & 32'h80000000))begin
    update_curr_mask = 1;          //a new longest match has been found
  end
  //entire table has been searched (4095 entries)
  if(index == 32'h4004) begin
    next_state = IDLE;
    FCMAPUDONE =1;
    FCMAPURESULTVALID =1;
  end
end
```

Software Application

Overview

To demonstrate the capabilities of this reference design, a software application was developed to utilize as many aspects of the design as possible. The resulting application creates a routing table in DDR memory and a similar table in block RAM. It then performs 64K LPM lookups on the table stored in DDR memory. The same 64K lookups are then performed using the hardware accelerator and the table stored in block RAM. Finally, the 64K lookups are again performed using the APU to Block RAM memory controller without hardware acceleration.

The routing tables are created in essentially the same manner as described in "Routing Table," page 2. When performing the lookups, every possible IP address between `0xFFFF0001` and `0xFFFFFFFF` is searched for. These are searched for in a pseudorandom but non-repeating order using the software LFSR to determine the next IP address to be looked up. The application is designed such that every one of these searches yields a valid result. As a check, before each block of 64K searches, 11 specific searches are done. The user is informed which searches pass and which ones fail as well as what data is stored at the index returned by the search. This information is printed to the UART. Of these 11 searches, 2 and 8 should not be found, while all others should be found. After completing all searches, the UART, operating at 115,200 baud, shows an output similar to the included file `proper_results.txt`.

The first number before each found/not found result is the time in microseconds that the search took to complete. The subsequent line is the IP address, mask, next hop, valid_port, and the address associated with the index returned by the search. At the end of the 11 searches, a line is printed with the memory access scheme being used (DDR, APU accelerated, or APU) along with statistics of the performance of the 64K lookups. All times are listed in microseconds.

Display

In addition to the debug information sent to the UART, a visual presentation of the data is displayed on a VGA screen using the TFT controller. The display shows two graphs:

- The left graph represents the 64K searches in the DDR-based table
- The right graph represents the 64K searches of the hardware-accelerated APU coprocessor

Because both searches look for every IP address between `0xFFFF0001` and `0xFFFFFFFF`, each of the 64K searches are represented by a single pixel on a 256 x 256 graph. The two highest bytes of the search key are always `0xFFFF`, but the two lower bytes can take on any

value. Therefore, the lowest byte of the current search key is used as the value along the y-axis, and the next byte is used as the value along the x-axis. Each time a search is performed, a pixel is placed on the graph in the location based on these lower two bytes of the IP address. The IP addresses are searched for in a pseudorandom order, so the graphs fill in a seemingly random order. When all 64K searches are done, the graph should be completely filled, with the exception of the pixel at (0,0) because this value is not searched for. The time taken to fill the DDR graph compared to the hardware-accelerated APU coprocessor graph is significant—the DDR memory searches take over 10 times longer.

Finally, the APU-connected memory is searched, and the performance is displayed. The results of this search are not graphed.

Key statistics of the three searches are shown when the search completes. After the three searches are finished, the searches repeat. During each search, a small green square is displayed next to the name of the search type being performed. After all three search types have been performed, the statistics remain but the graphs are cleared before the searches begin again. The total lookup times between 64K searches show very minor variations. However, the average lookup time for an individual search of any type remains constant.

Performance

Overview

In addition to proving feasibility of the design, this project also evaluated performance against alternative memory-access methods. The LPM algorithm was kept constant while the memory type used to store the data and the amount of custom hardware versus software were varied. The routing table was implemented in APU-attached block RAM, PLB-attached DDR memory, PLB-attached block RAM, and OCM-attached block RAM. The physical constraints of the Virtex-4 device limit the total amount of block RAM to 72 KB. Thus, it is not possible to implement a 64 KB OCM-attached block RAM or a PLB-attached block RAM design at the same time as a 64 KB APU-attached block RAM version of the design. Separate systems were generated to obtain comparative numbers for OCM, DDR, and APU designs. The APU-based method of storing the routing table can optionally use the hardware as an accelerating coprocessor. The LPM searches that were benchmarked are:

- PLB-attached DDR memory
- PLB-attached block RAM
- OCM-attached block RAM
- APU-attached block RAM
- Hardware-accelerated APU coprocessor

This project focused on the two APU-based lookup methods. The APU-attached block RAM was expected to be slower than the OCM-attached block RAM, while having superior performance to the PLB-attached DDR memory. This was demonstrated. However, the dead cycle required after each APU instruction added overhead, reducing performance. The hardware-accelerated APU coprocessor achieved speeds that were an order of magnitude faster than all the other methods.

Data

The raw lookup times for an LPM search are listed in Table 6 for each method of storing and accessing the routing table. Statistics for single lookups are provided as well as the total time taken by all 64K searches.

Table 6: **Raw Lookup Times for LPM Search (µs)**

Type	Average Successful (µseconds)	Total Time for 64K Searches (seconds)	Relative Speed
PLB DDR Memory	839	55.3	1.0
PLB Block RAM	398	27.0	2.0
OCM Block RAM	325	22.2	2.5
APU Block RAM	529	34.8	1.6
Hardware Accelerated APU Block RAM	41	2.7	20.5

Notes:

1. Main processor clock = 200 MHz, PLB clock = 100 MHz, FCM clock = 100 MHz.
2. PLB DDR memory performance might be improved with increased PLB clock rates.

Reference Design

Required Hardware and Software Tools

The required tools for this reference design are:

- Xilinx ML403 Virtex-4 evaluation platform with power supply
- JTAG programming cable connected to the ML403 and PC
- VGA monitor with VGA cable connected to the ML403
- Serial null-modem female-to-female cable connected between the ML403 and PC
- Xilinx ISE™ tool, v9.2.03i or later
- Xilinx Platform Studio, v9.2.02i or later

Full Design Device Utilization Summary

Table 7 summarizes the utilization of the devices in the reference design.

Table 7: **Reference Design Device Utilization Summary**

Resource Type	Used	Available	Utilization (%)
BUFG	4	32	12
DCM_ADV	1	4	25
ILOGIC	33	320	10
External IOB	88	320	27
LOCed IOB	88	94	100
JTAGPPC	1	1	100
OLOGIC	85	320	26
PPC405_ADV	1	1	100
RAMB16	36	36	100
Slice	4302	5472	78
SLICEM	478	2736	17
4-input LUT	5186	10944	47

32 RAMB16s are used for the APU's routing table. The TFT controller also uses a single RAMB16.

Table 8 shows the project files used in the reference design. The files are extracted to the `Xilinx_Design\V4FX_Labs\LPM` directory. They are available for download at https://secure.xilinx.com/webreg/clickthrough.do?cid=103956.

Table 8: **Reference Design Project File Utilization Summary**

Filename	Description
`readme.txt`	Text file explaining the reference design and how to use it.
`demo_download.bit`	Pregenerated bitstream file ready for download.
`proper_results.txt`	Sample UART output for a successful run.
`..\pcores\apu_to_bram_v1_00_a\hdl\verilog\apu_to_bram.v`	Verilog source file of hardware used in the APU coprocessor.
`..\pcores\apu_to_bram_v1_00_a\hdl\vhdl\apu_to_bram.vhd`	VHDL source file of hardware used in the APU coprocessor.
`..\LPM\src\bootload_basicgraphics.c`	Basic functions to plot on the TFT display.
`..\LPM\src\bootload_basicgraphics.h`	Header for `bootload_basicgraphics.c`.
`..\LPM\src\initializeDisplay.c`	Contains methods to display data on the TFT display.
`..\LPM\src\initializeDisplay.h`	Header for `initializeDisplay.c`.
`..\LPM\src\LPM.c`	Performs the longest prefix match algorithm. Provides functions used in filling, searching, and printing entries from a table.
`..\LPM\src\LPM.h`	Header for `LPM.c`.
`..\LPM\src\lpm_main.c`	Contains the main method. Demonstrates the lookup times of different memory storage and access methods.
`..\LPM\src\stop_watch.c`	Timer functions.
`..\LPM\src\stop_watch.h`	Header for `stop_watch.c`.
`..\LPM\src\LPM_LinkScr.ld`	Custom linker script.

Figure 7 shows a screenshot of the application.

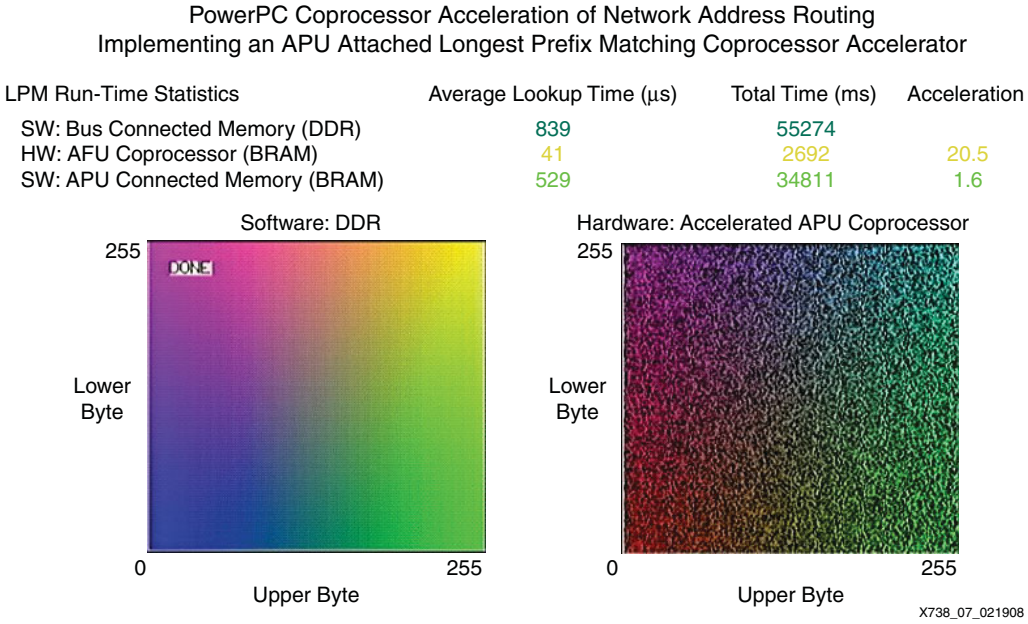


Figure 7: **Screenshot of Application**

Conclusion

The APU interface can be a preferable method of communicating with memory. As the data shows, even with a relatively small table size of 64 KB, the APU-attached block RAM exceeds the performance of the PLB-attached DDR memory. The APU-attached block RAM is better suited than the PLB-attached DDR memory for designs with larger table sizes where less of the table is cacheable. Although the OCM-attached block RAM is faster than APU-attached block RAM, the difference in performance is relatively small. Additional benefits to using the APU interface include the ability to add a coprocessor, if needed, and to continue issuing CPU instructions while autonomous APU instructions execute in parallel.

There are significant benefits to adding custom hardware to the design attached via the APU interface. Even the speed of a simple C routine used to perform a lookup can be increased by an order of magnitude by implementing custom acceleration hardware. Attaching block RAM and custom hardware via the same APU interface enables flexibility in both the operation and organization of the block RAM. The ability to read and write 32-bit values from memory and alternatively have a coprocessor read 128 bits from memory during a lookup illustrates this flexibility in working with the APU interface.

Acknowledgment

Xilinx wishes to thank Tristan Johnson for his contributions to this reference design and document.

Revision History

The following table shows the revision history for this document:

Date	Version	Description of Revisions
02/22/08	1.0	Initial Xilinx release.

Notice of Disclaimer