FSM

In this lecture, we begin looking at the options for mapping algorithms into hardware

We begin at the behavioral level of abstraction with a **register-transfer-level** (RTL) description of hardware using VHDL

At the heart of RTL descriptions is the *finite state machine with datapath* (**FSMD**)

The *finite state machine* component is constructed (usually by the synthesis tools) to realize the *control flow* components of the algorithm

The *datapath* is synthesized into a set of mathematical hardware operations, e.g., add and multiply, that process the data

We begin with algorithms that are largely characterized as control-only algorithms

- We consider full-blown FSMD in future lectures

Our goal is to become proficient at translating software algorithms (and other high-level specifications) into hardware implementations

FSM

An FSM is designed to control the execution of a sequence of operations over multiple clock cycles, which makes it ideal for emulating software execution

It usually incorporates decision constructs, including *if-else* and *case* statements, which are used extensively in software execution for implementing control flow

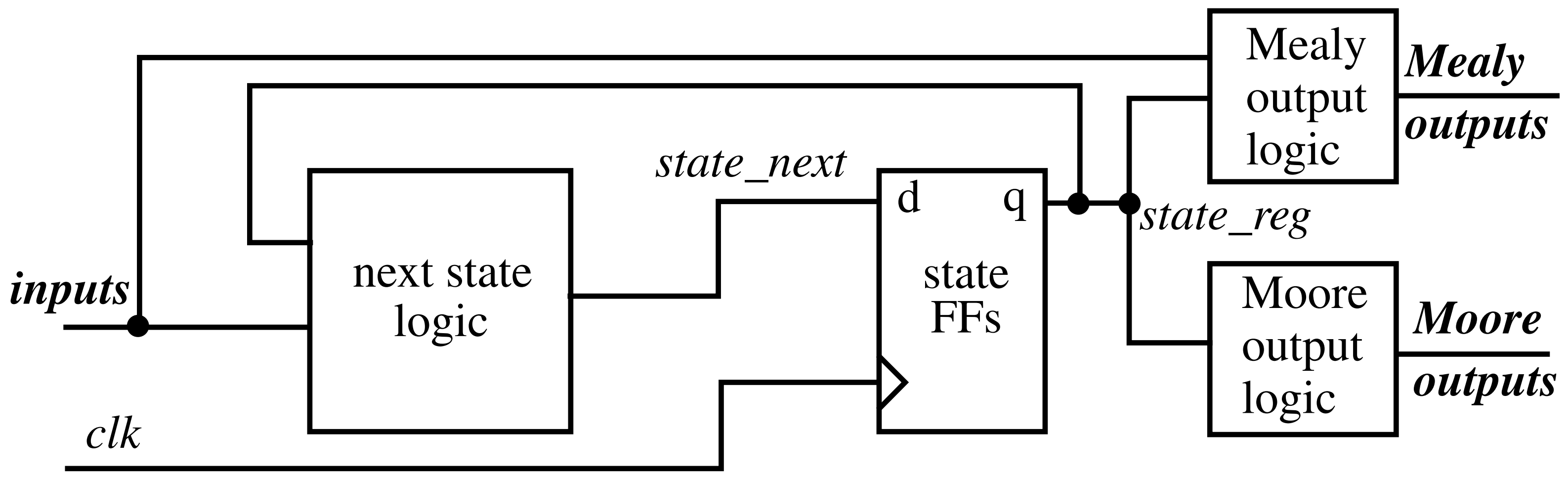
Advanced FSMs can be designed to implement pipelines, recursion, out-of-order execution and exception handling

An FSM is a sequential digital machine, which is defined using:

- A set of states
- A set of inputs and outputs
- A state transition function
- An output function

FSM

The Mealy form of an FSM (as you've seen in previous classes) allows the *inputs* to effect both the *next state logic* and *outputs*



Both Mealy and Moore outputs can be present in an FSM, and usually are

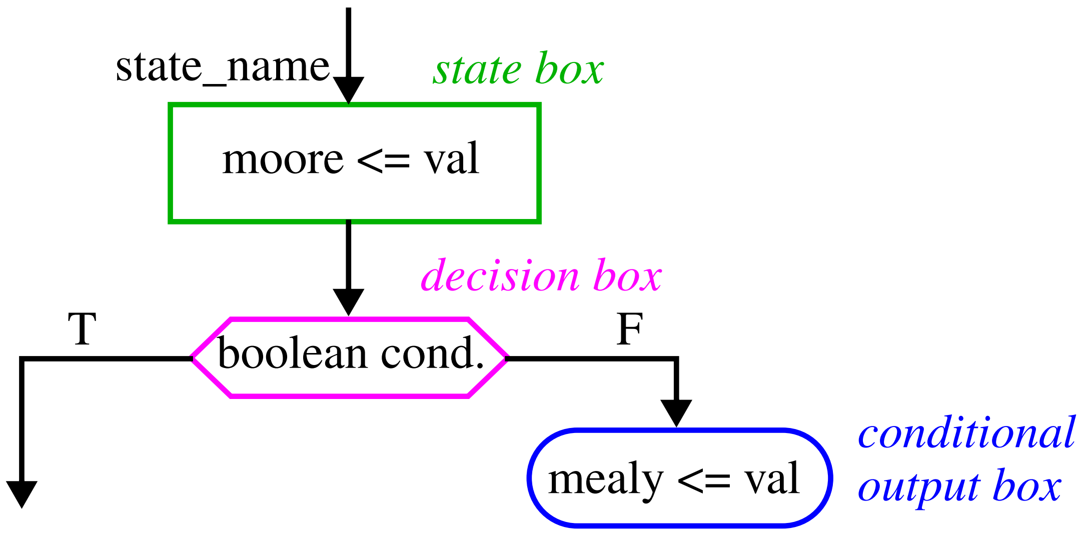
Moore outputs depend only on the current state

VHDL descriptions of the FFs and combinational logic (*next state logic* and *output logic*) are separated into two **process** blocks using two segment style

ASM

State diagrams are the most common graphical representation of FSMs, but algorithmic state machine (ASMs) diagrams can also be used

ASMs provide a more explicit representation of control and data path elements



Each *state box* has only one exit and is usually followed by a *decision box*

Conditional output boxes can only follow *decision* boxes and are used to define the values of Mealy outputs as a function of the Boolean conditions

EVERYTHING that follows a state box (to the next state) is **combinational logic** that is *active* in the current clock cycle

Secure Memory Access Controller

Let's consider a control algorithm that is designed to provide a secure access control mechanism to an on-chip memory

We discussed the architecture and provided instruction on how to create a project with **GPIO** and **BRAM** IP blocks in the laboratory screencasts

The general purpose I/O is configured to implement two memory-mapped 32-bit registers located between the PS and PL sides of the Zynq SoC

The BRAM is configured as a stand-alone memory and is embedded within the PL side

Our memory controller's primary function is to allow C programs running on the ARM microprocessor to load and unload the BRAM in a restricted manner

High Level Algorithm for a Secure Memory Access Controller

Let’s start with at level of abstraction of the FSM running in the hardware, e.g.,

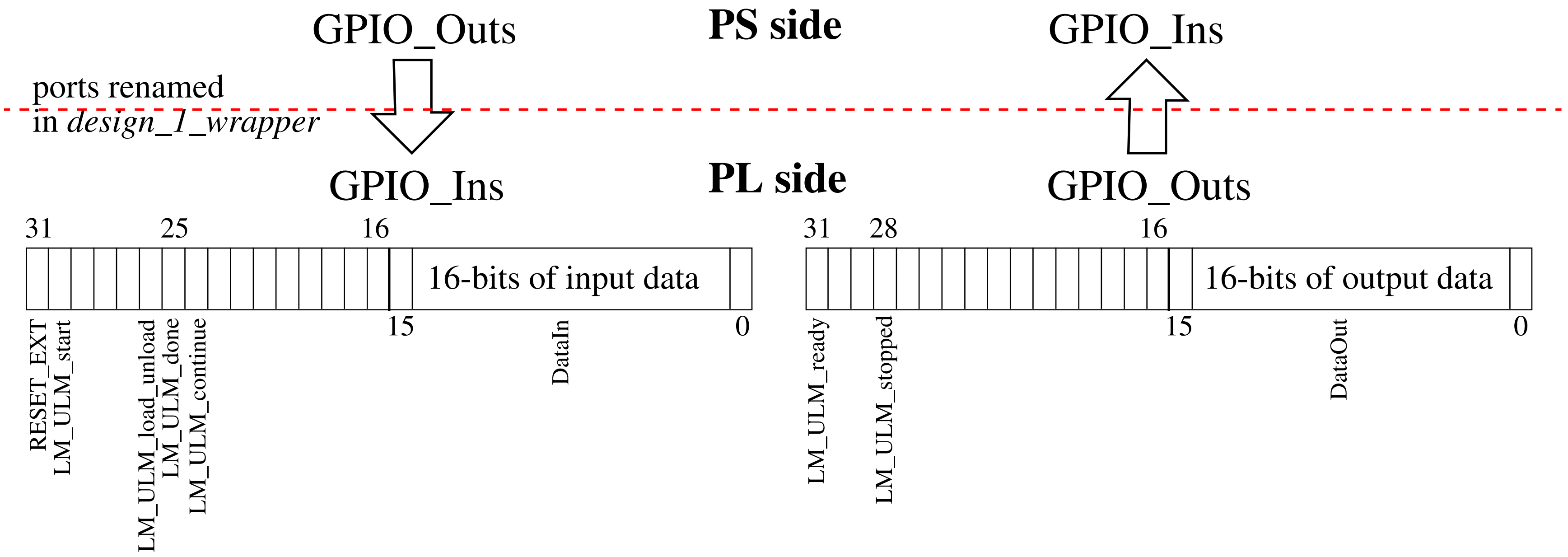
```
if ( 'start' == 1 )
   store 'base addr' and 'upper limit'
   while ('base addr' != 'upper limit' and 'done' == 0 )
      if ( 'load_unload' == 0 )
         PNL_BRAM['base addr'] = GPIO
      else
         GPIO = BRAM_PNL['base addr']
      'base addr' = 'base addr' + 1
```

One of the first issues we’ll need to deal with is setting up a communication mechanism between the C program and the FSM

The GPIOs are visible to both the C program and the VHDL, and will be used for this purpose

GPIO Register Definitions for Secure Memory Access Controller

We will define the bit-fields within the two 32-bit GPIO registers as follows:



Some of the high order 16-bits are designated for **control** while all of the lower order 16-bits are designated for **data** transfers

Note that the GPIOs cross clock domains, with the PL side running at 50 or 100 MHz and the PS side running at more than 600 MHz

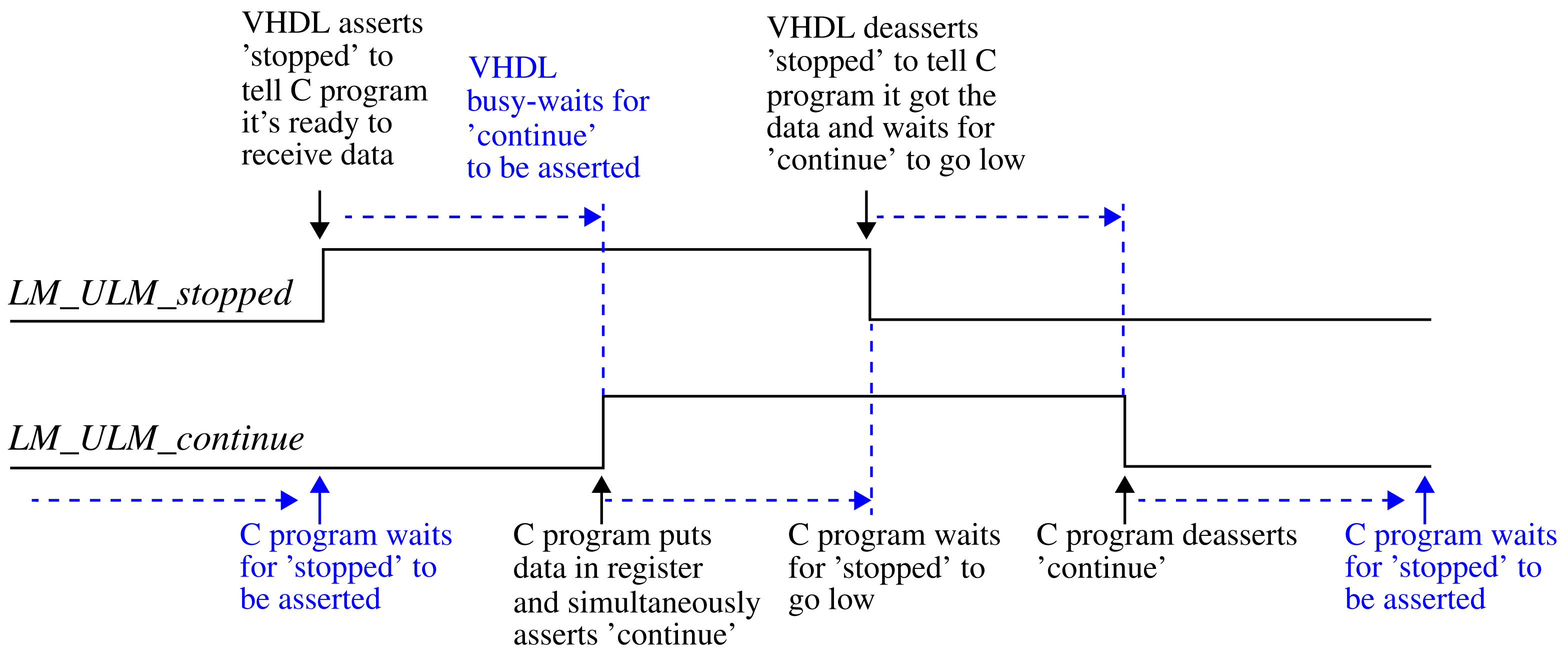
Therefore, we need a reliable protocol to allow transfers between PL and PS

HandShake Synchronization for Secure Memory Access Controller

Synchronization between our VHDL controller (PL) and the C program (PS) is done using a 2-way handshake, which makes use of two signals

LM_ULM_stopped and *LM_ULM_continue*

The following defines the protocol for data transfers **from** the C program **to** BRAM:


VHDL asserts 'stopped' to tell C program it's ready to receive data
VHDL busy-waits for 'continue' to be asserted
VHDL deasserts 'stopped' to tell C program it got the data and waits for 'continue' to go low
LM_ULM_stopped
LM_ULM_continue
C program waits for 'stopped' to be asserted
C program puts data in register and simultaneously asserts 'continue'
C program waits for 'stopped' to go low
C program deasserts 'continue'
C program waits for 'stopped' to be asserted

Data transfer from BRAM to C program is similar except for direction of data flow

This protocol ensures a reliable communication channel between the PS and PL side

Low Level Algorithm for Secure Memory Access Controller

We are now ready to describe the FSM at a lower level of detail

Some parts of the following pseudo-code may seem irrelevant, but in fact will make the algorithm more versatile

For example, we provide a *done* flag that the C program will assert to allow it to terminate the memory operations

In fact, *done* will allow the C program to terminate before doing any reads or writes whatsoever!

In our version, the C program starts the secure memory access controller (SMAC) FSM, but other usage scenarios may have other state machines start SMAC

In this case, the C program may need to inform SMAC that it has no read or write requests for the memory at that point in time

The C program operations in the following are designed to provide context, and are not part of the FSM

Algorithm for Secure Memory Access Controller

```
1) C program checks 'ready', sets 'load_unload' flag and
issues 'start' signal
2) C program waits for 'stopped'
3) idle: SMAC waits in idle for 'start'
   if ( start = '1' )
      Store 'base address' and 'upper limit' registers
      Check 'load_unload', if 0, Goto load_mem
      Check 'load_unload', if 1, Goto unload_mem
4) load_mem: Process write requests from C program
   Assert 'stopped' signal
   If 'done' is 0
      Check 'continue', if asserted, update BRAM
         Assert 'PNL_BRAM_we'
         Assign 'PNL_BRAM_din' GPIO data
         Goto wait_load_unload
   else
      Goto wait_done
```

Algorithm for Secure Memory Access Controller

```
5) unload_mem: Process read requests from C program
   Drive GPIO register with 'PNL_BRAM_dout' data
   Assert 'stopped'
   If 'done' is 0
      Check 'continue', if asserted, C program has data
      Goto wait_load_unload
   else
      Goto wait_done
6) wait_load_unload: Finish handshake and update addr
   Check 'continue', if 0
      If 'done', if 1
         Goto wait_done
      Elsif 'base addr' = 'upper_limit'
         Goto idle
      Else
         Inc 'base addr'
         Goto load_mem if 'load_unload' is 0
         Goto unload_mem if 'load_unload' is 1
```

Algorithm for Secure Memory Access Controller

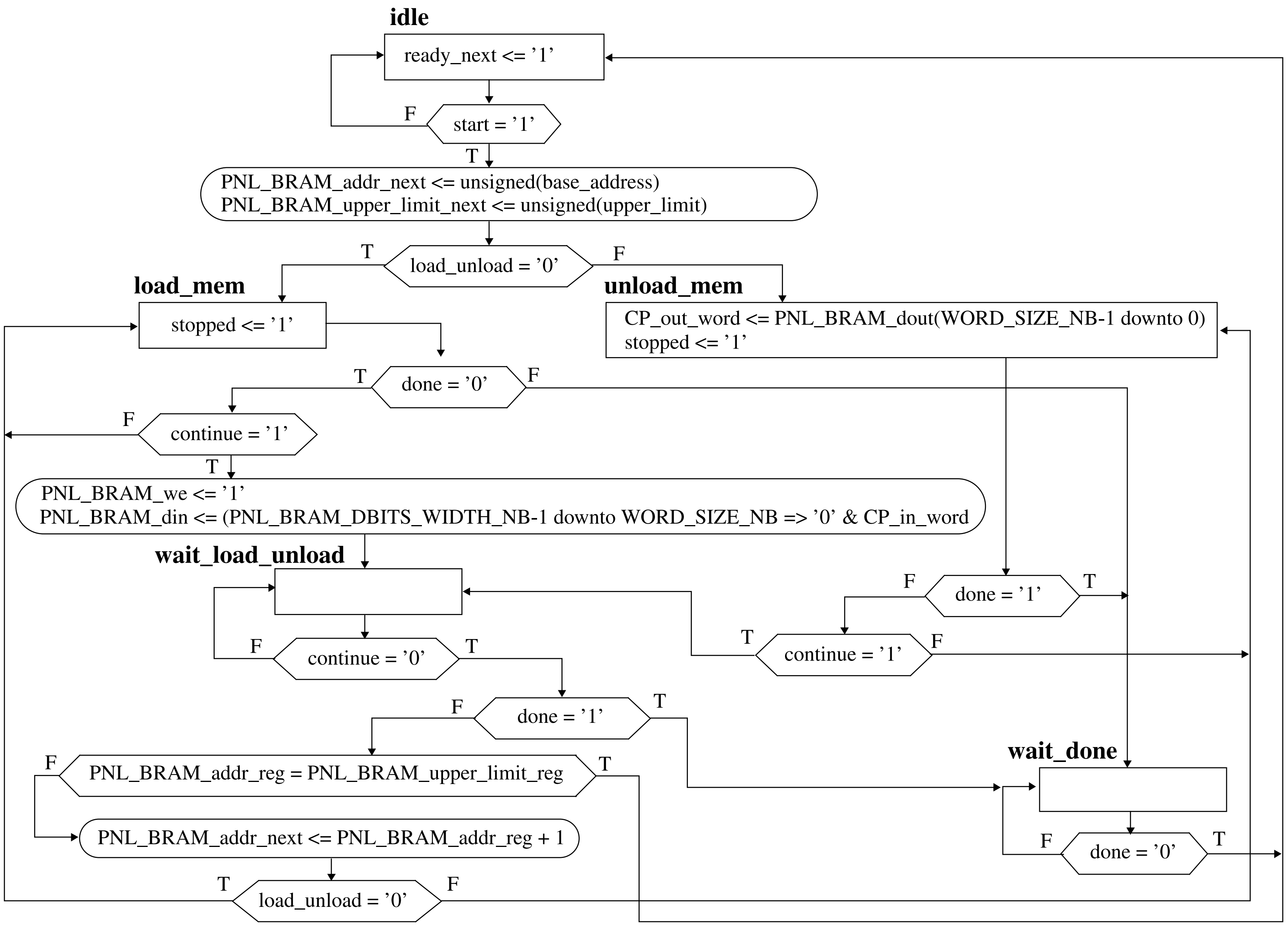
```
7) wait_done: Wait for C program to deassert 'done'
   Check 'done', if 0
      Goto idle
```

I was able to structure the pseudo-code directly into a form compatible with an FSM

As you can see, the lower level of abstraction has much more detail, with a 'goto-like' structure to implementing the state transition diagram

It also reveals elements of control in hardware design not found in software design, e.g., the 'write enable' (PNL_BRAM_we) associated with the memory

ASM For LoadUnloadMem.vhd


idle
ready_next <= ’1’
F
start = ’1’
T
PNL_BRAM_addr_next <= unsigned(base_address)
PNL_BRAM_upper_limit_next <= unsigned(upper_limit)
T
load_unload = ’0’
F
load_mem
stopped <= ’1’
unload_mem
CP_out_word <= PNL_BRAM_dout(WORD_SIZE_NB-1 downto 0)
stopped <= ’1’
T
done = ’0’
F
F
continue = ’1’
T
PNL_BRAM_we <= ’1’
PNL_BRAM_din <= (PNL_BRAM_DBITS_WIDTH_NB-1 downto WORD_SIZE_NB => ’0’ & CP_in_word
wait_load_unload
F
done = ’1’
T
F
continue = ’0’
T
T
continue = ’1’
F
F
done = ’1’
T
wait_done
F
PNL_BRAM_addr_reg = PNL_BRAM_upper_limit_reg
T
PNL_BRAM_addr_next <= PNL_BRAM_addr_reg + 1
F
done = ’0’
T
T
load_unload = ’0’
F