N26F300
VLSI SYSTEM DESIGN
(GRADUATE LEVEL)

Fall 2010

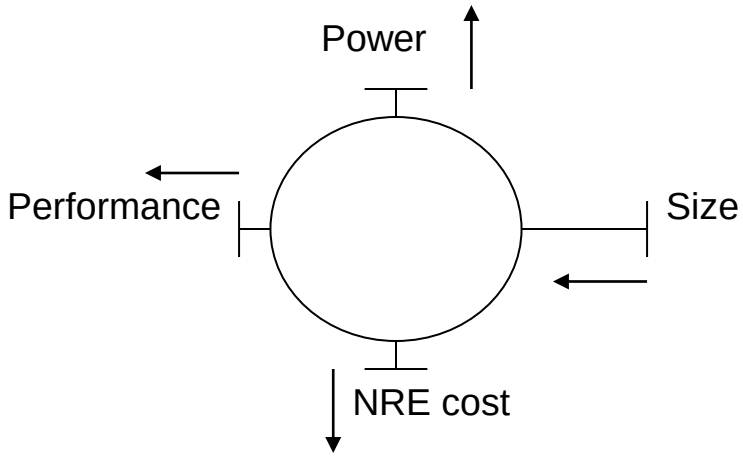
Processors & Peripherals & Interfacing

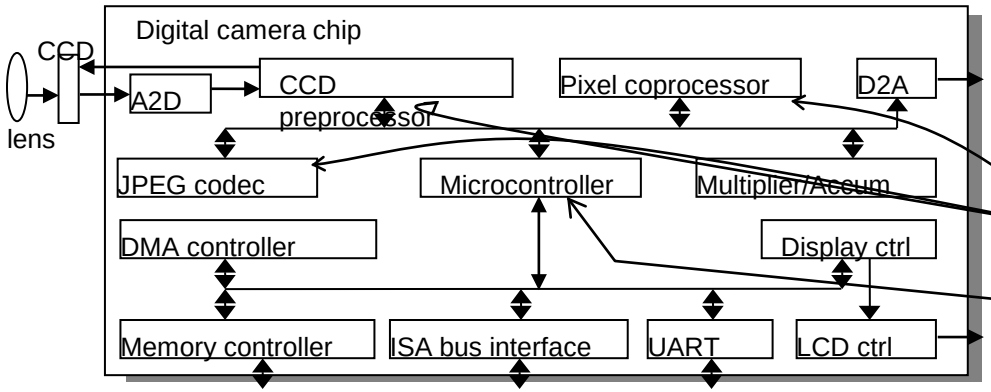
Outline

- Processor
- Custom processor – GCD example
- Peripherals
- Interfacing via bus

[Material partly adapted from Embedded System Design by F. Vahid & T. Givargis]

Design metric competition -- improving one may worsen others

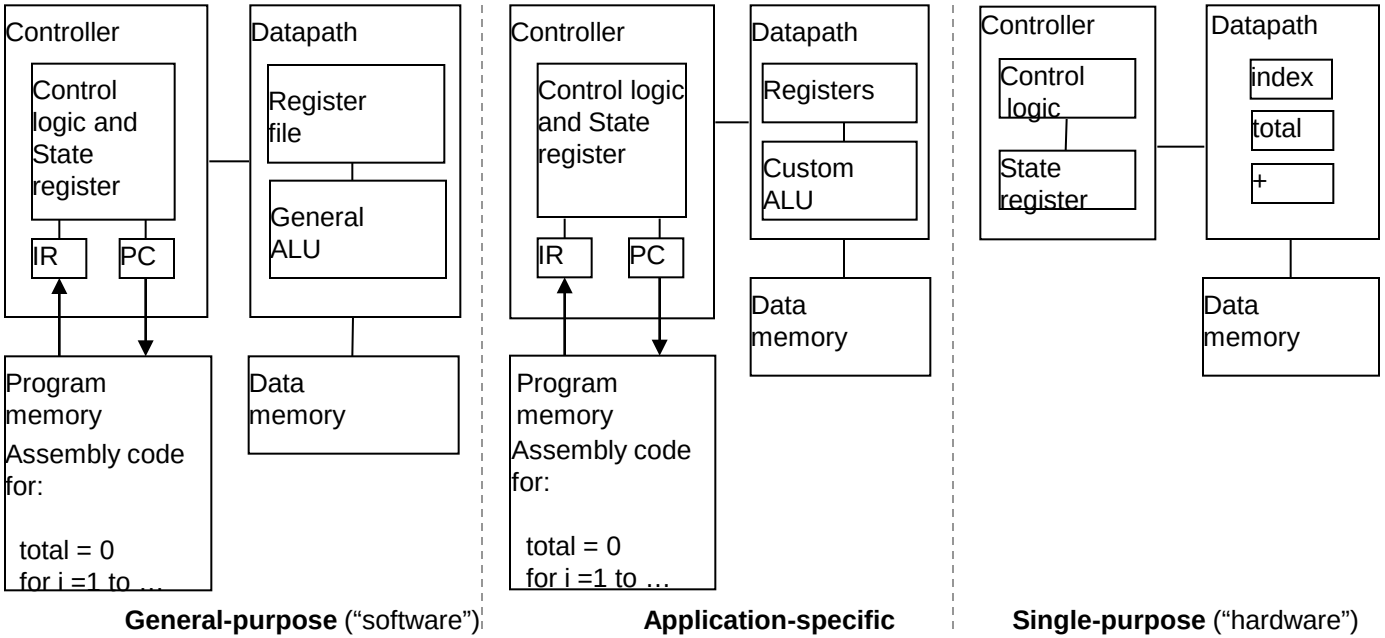




- Expertise with both **software and hardware** is needed to optimize design metrics
  - Not just a hardware or software expert, as is common
  - A designer must be comfortable with various technologies in order to choose the best for a given application and constraints

Processor technology

- The architecture of the computation engine used to implement a system's desired functionality
- Processor does not have to be programmable
  - "Processor" *not* equal to general-purpose processor



Processor technology

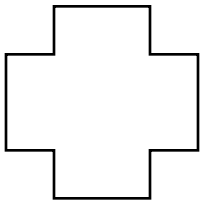
- Processors vary in their customization for the problem at hand

Desired functionality

```
total = 0
for i = 1 to N  loop
   total += M[i]
end loop
```

General-purpose processor

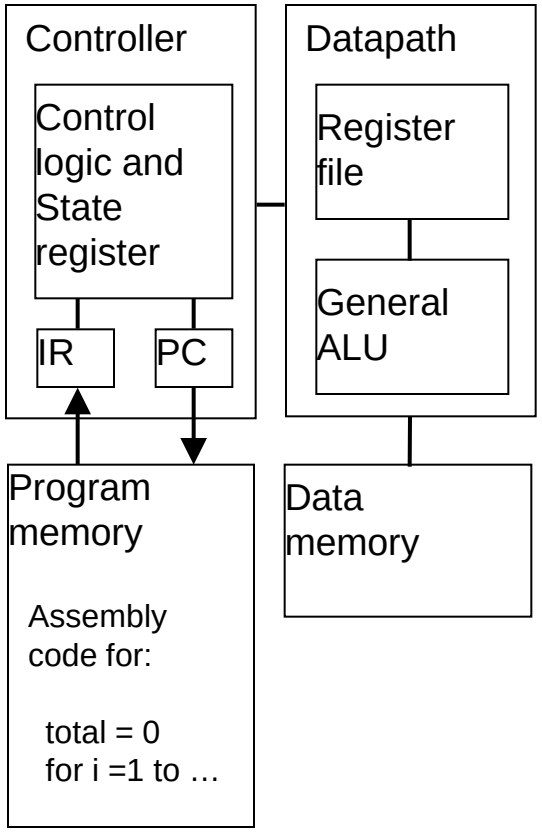
Application-specific processor



Single-purpose processor

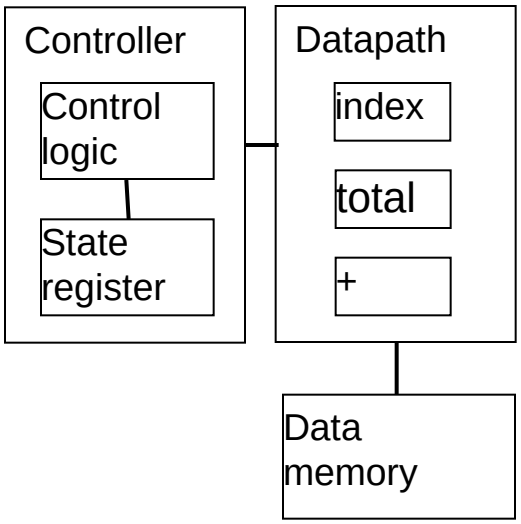
General-purpose processors

- Programmable device used in a variety of applications
  - Also known as “microprocessor”
- Features
  - Program memory
  - General datapath with large register file and general ALU
- User benefits
  - Low time-to-market and NRE costs
  - High flexibility
- “Pentium” the most well-known, but there are hundreds of others



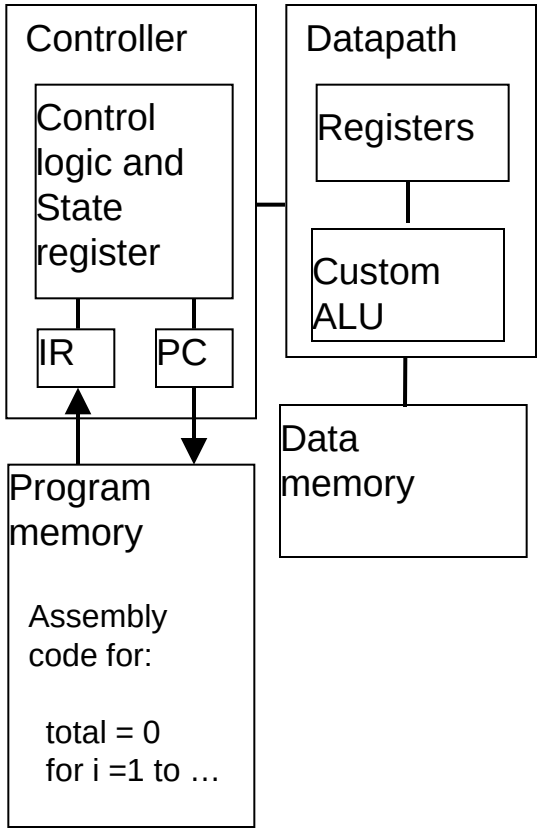
Single-purpose processors

- Digital circuit designed to execute exactly one program
  - a.k.a. coprocessor, accelerator or peripheral
- Features
  - Contains only the components needed to execute a single program
  - No program memory
- Benefits
  - Fast
  - Low power
  - Small size



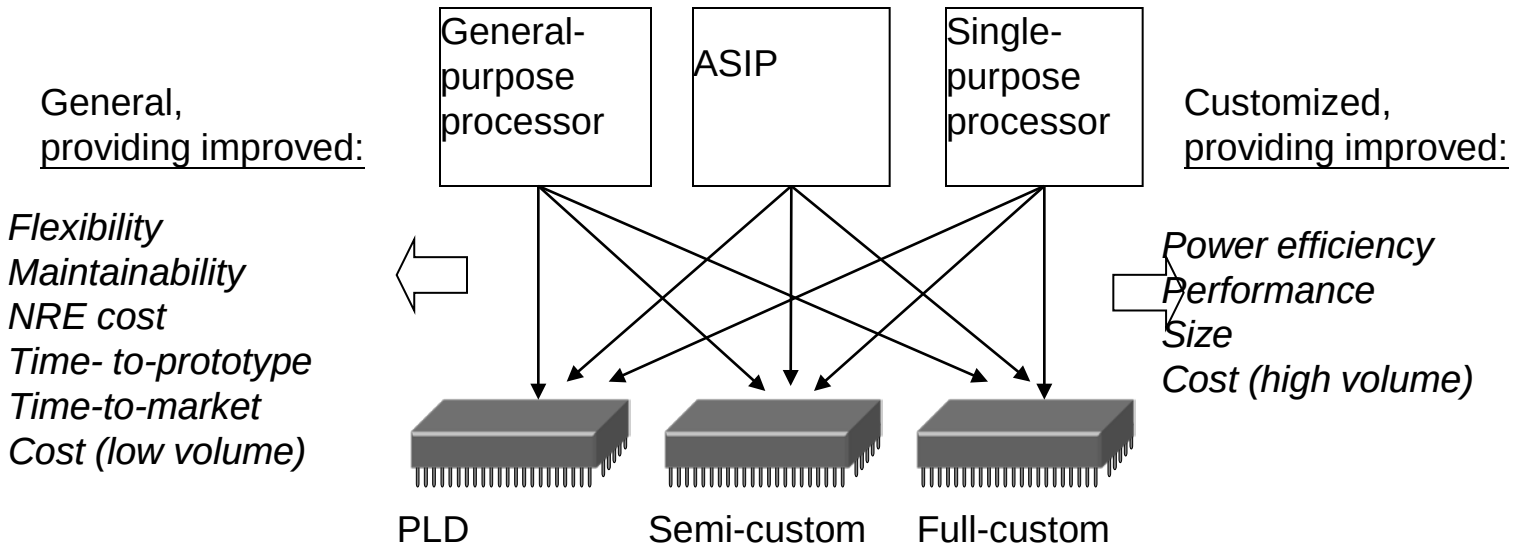
Application-specific processors

- Programmable processor optimized for a particular class of applications having common characteristics
  - Compromise between general-purpose and single-purpose processors
- Features
  - Program memory
  - Optimized datapath
  - Special functional units
- Benefits
  - Some flexibility, good performance, size and power



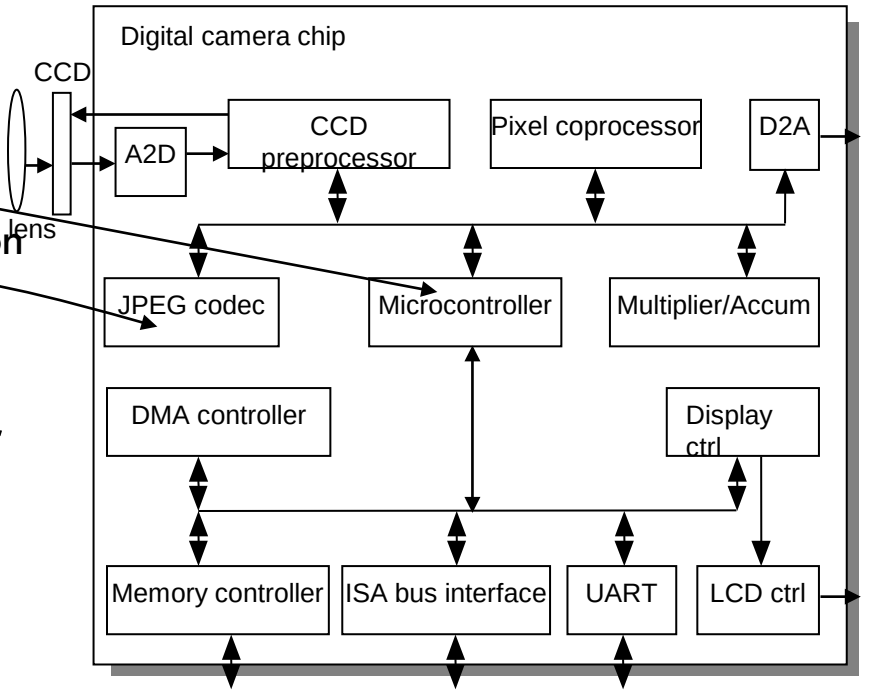
Independence of processor and IC technologies

- Basic tradeoff
  - General vs. custom
  - With respect to processor technology or IC technology
  - The two technologies are independent



Custom Processor

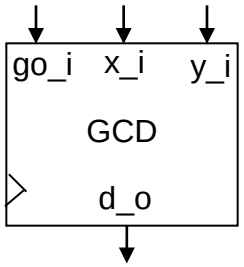
- Processor
  - Digital circuit that performs a computation tasks
  - Controller and datapath
  - General-purpose: variety of computation tasks
  - Single-purpose: one particular computation task
  - Custom single-purpose: non-standard task
- A custom single-purpose processor may be
  - Fast, small, low power
  - But, high NRE, longer time-to-market, less flexible



Example: greatest common divisor

- First create algorithm
- Convert algorithm to "complex" state machine
  - Known as FSMD: finite-state machine with datapath
  - Can use templates to perform such conversion

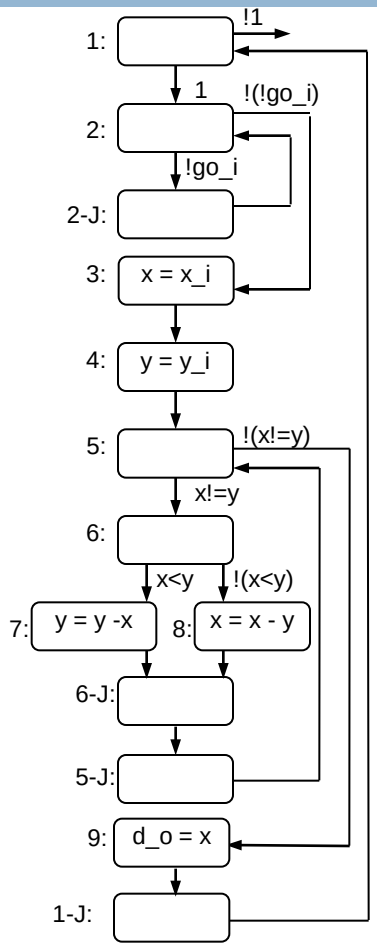
(a) black-box view



(b) desired functionality

```
0: int  x, y;
1: while (1) {
2:   while (!go_i);
3:   x = x_i;
4:   y = y_i;
5:   while  (x != y)  {
6:       if  (x < y)
7:           y = y - x;
         else
8:           x = x - y;
     }
9:   d_o = x;
   }
```

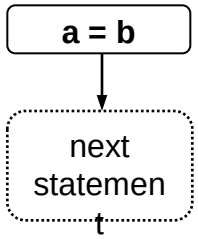
(c) state diagram



State diagram templates

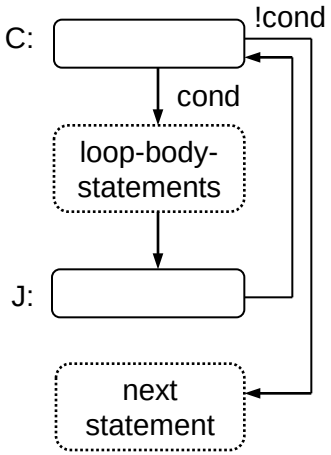
Assignment statement

a = b
next
statement



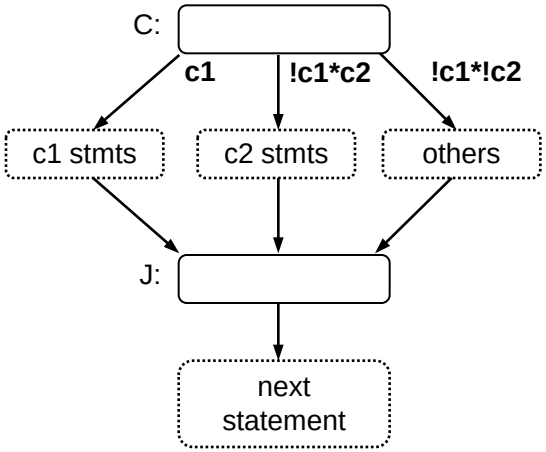
Loop statement

while (cond) {
loop-body-
statements
}
next statement



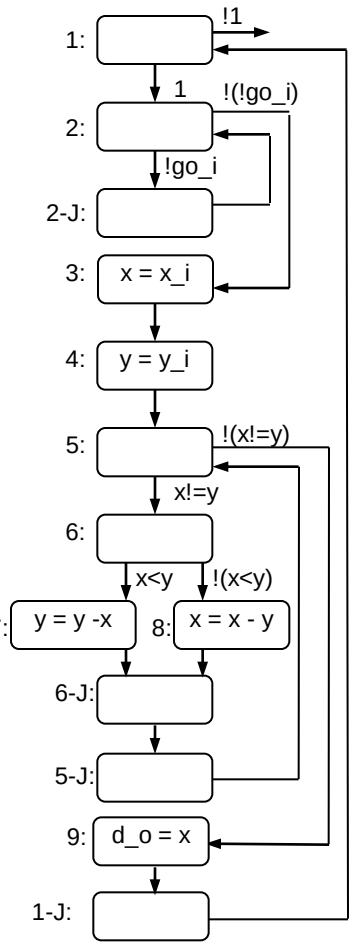
Branch statement

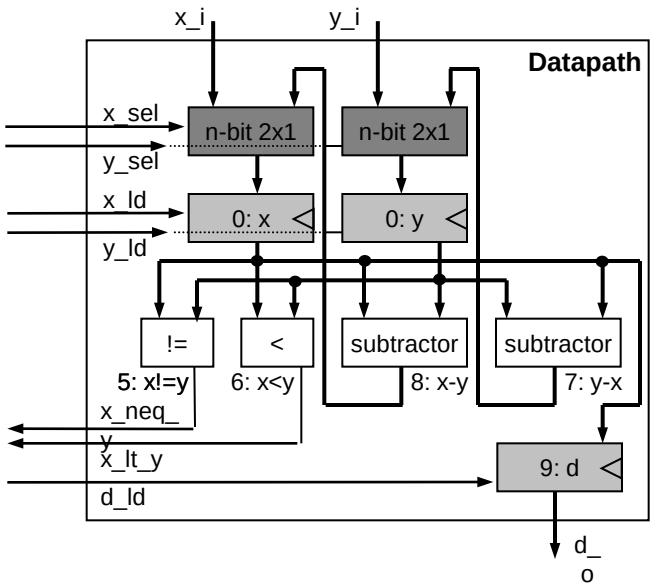
if (c1)
c1 stmts
else if c2
c2 stmts
else
other stmts
next statement



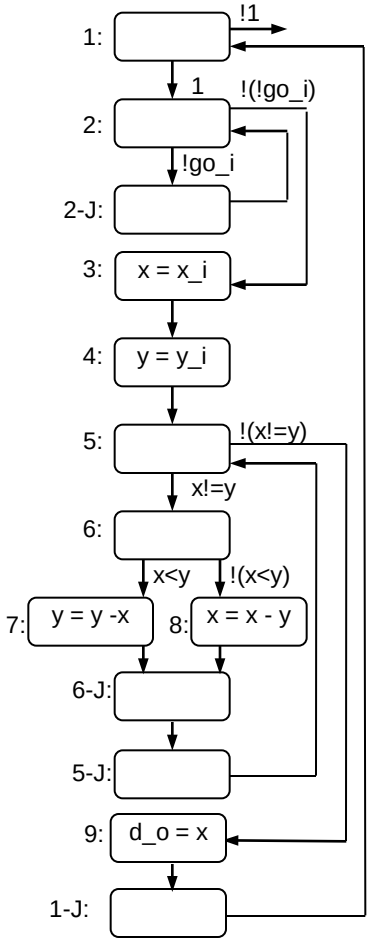
Creating the datapath

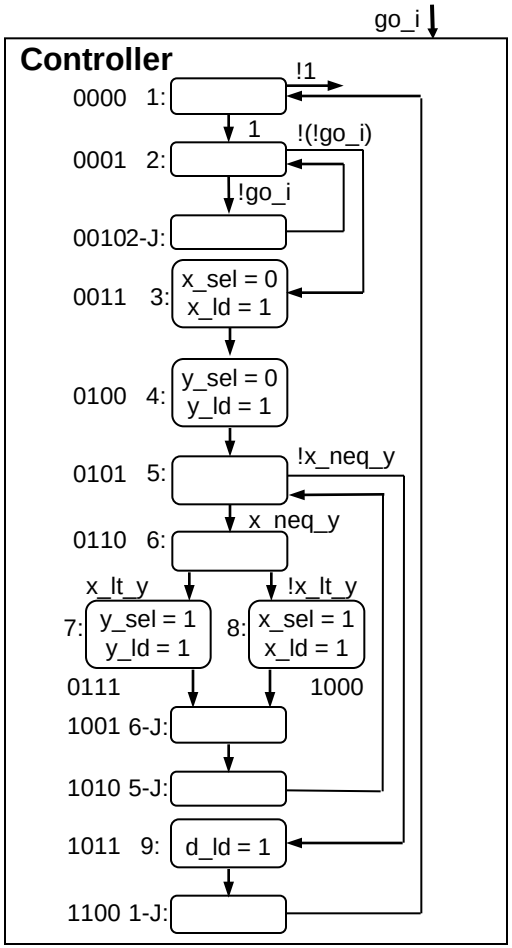
- Create a register for any declared variable
- Create a functional unit for each arithmetic operation
- Connect the ports, registers and functional units
  - Based on reads and writes
  - Use multiplexors for multiple sources
- Create unique identifier
  - for each datapath component control input and output



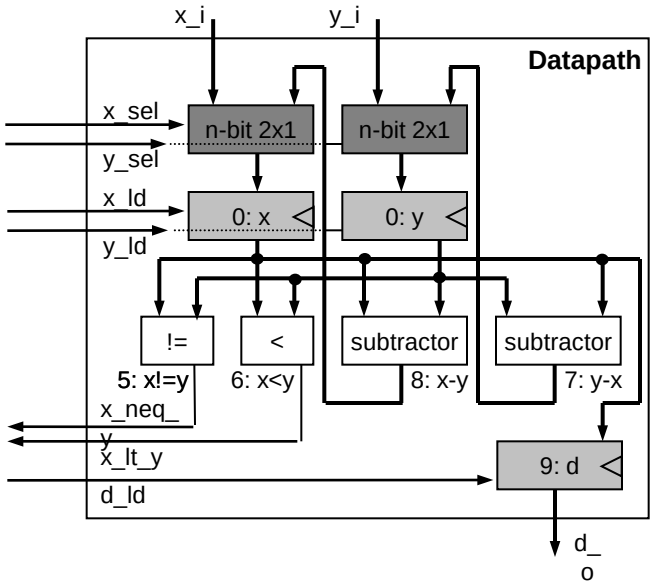


Creating the controller's FSM

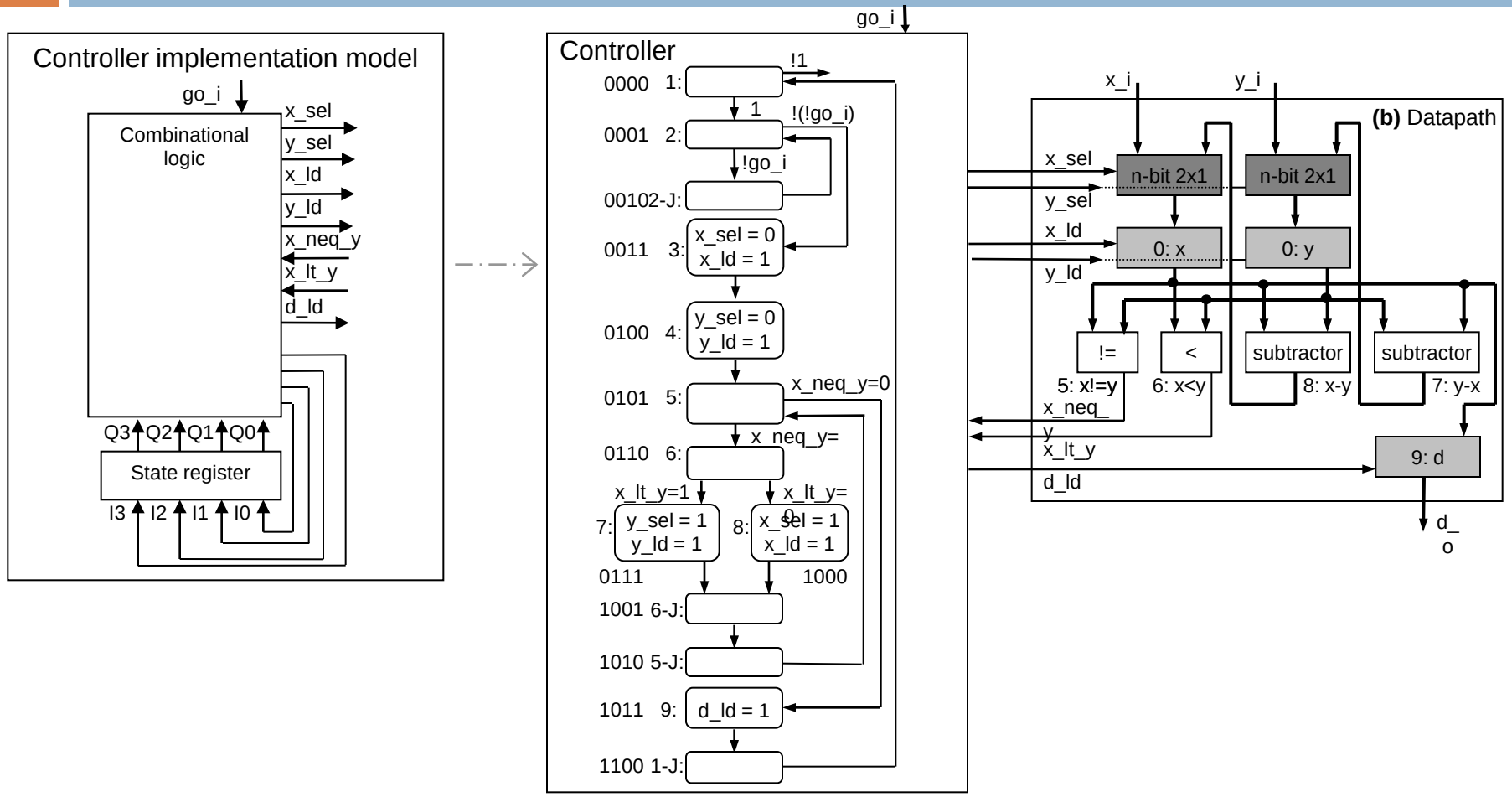




- Same structure as FSMD
- Replace complex actions/conditions with datapath configurations



Splitting into a controller and datapath

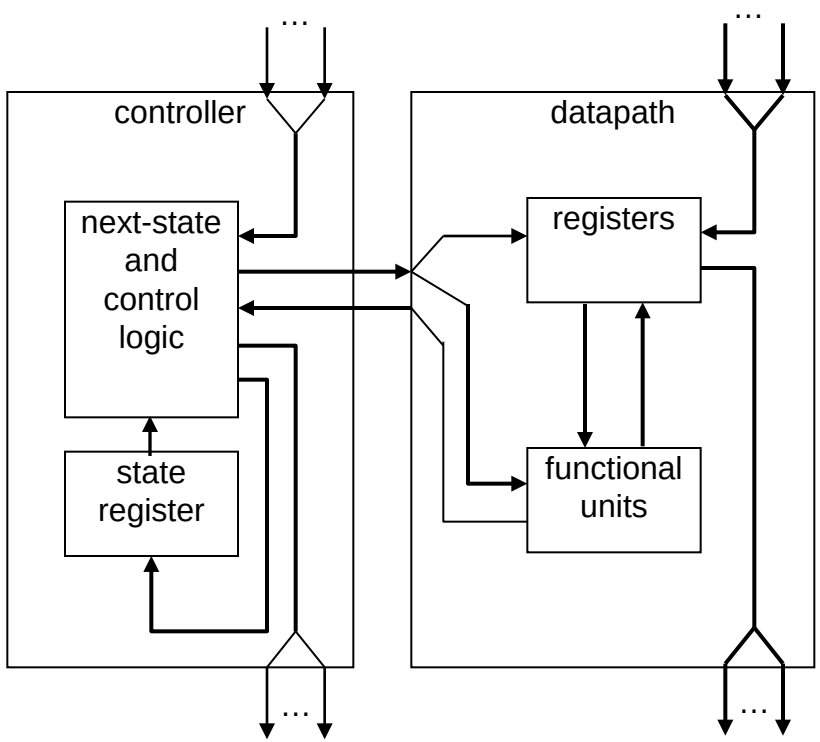


Controller state table for the GCD example

Inputs							Outputs								
Q3	Q2	Q1	Q0	x_neq_y	x_lt_y	go_i	I3	I2	I1	I0	x_sel	y_sel	x_ld	y_ld	d_ld
0	0	0	0	*	*	*	0	0	0	1	X	X	0	0	0
0	0	0	1	*	*	0	0	0	1	0	X	X	0	0	0
0	0	0	1	*	*	1	0	0	1	1	X	X	0	0	0
0	0	1	0	*	*	*	0	0	0	1	X	X	0	0	0
0	0	1	1	*	*	*	0	1	0	0	0	X	1	0	0
0	1	0	0	*	*	*	0	1	0	1	X	0	0	1	0
0	1	0	1	0	*	*	1	0	1	1	X	X	0	0	0
0	1	0	1	1	*	*	0	1	1	0	X	X	0	0	0
0	1	1	0	*	0	*	1	0	0	0	X	X	0	0	0
0	1	1	0	*	1	*	0	1	1	1	X	X	0	0	0
0	1	1	1	*	*	*	1	0	0	1	X	1	0	1	0
1	0	0	0	*	*	*	1	0	0	1	1	X	1	0	0
1	0	0	1	*	*	*	1	0	1	0	X	X	0	0	0
1	0	1	0	*	*	*	0	1	0	1	X	X	0	0	0
1	0	1	1	*	*	*	1	1	0	0	X	X	0	0	1
1	1	0	0	*	*	*	0	0	0	0	X	X	0	0	0
1	1	0	1	*	*	*	0	0	0	0	X	X	0	0	0
1	1	1	0	*	*	*	0	0	0	0	X	X	0	0	0
1	1	1	1	*	*	*	0	0	0	0	X	X	0	0	0

Completing the GCD custom single-purpose processor design

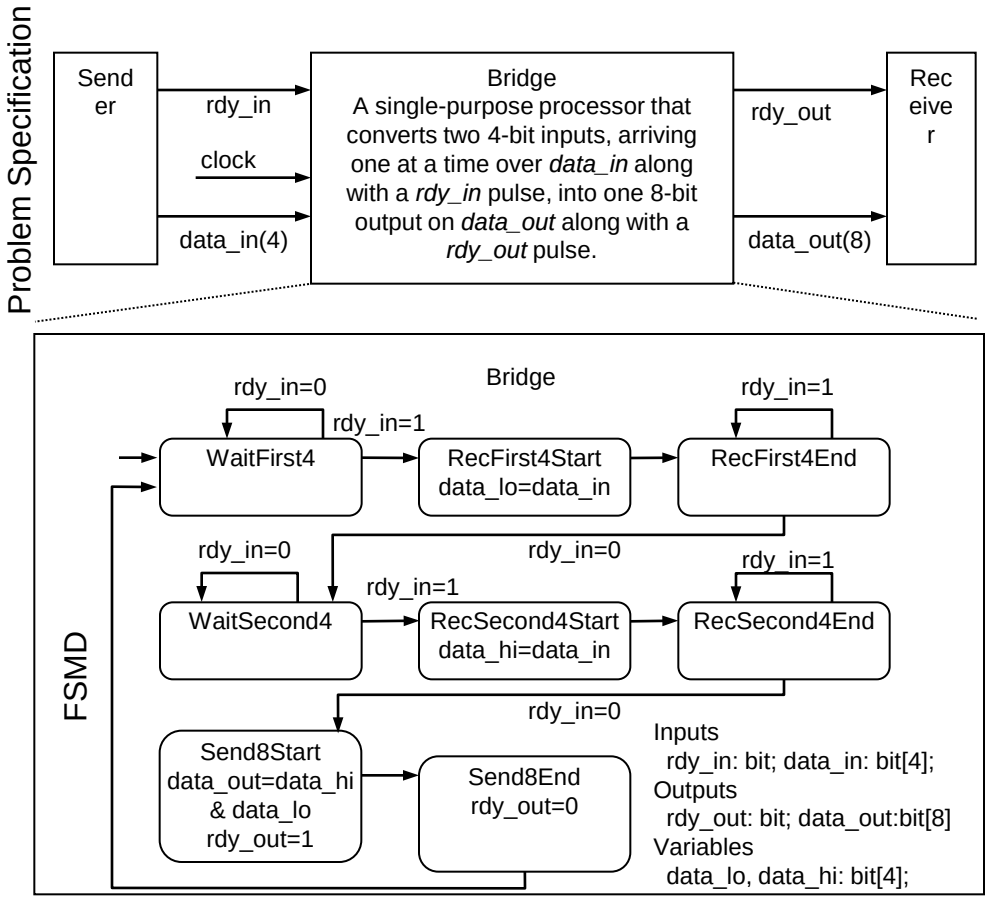
- We finished the datapath
- We have a state table for the next state and control logic
  - All that's left is combinational logic design
- This is *not* an optimized design, but we see the basic steps



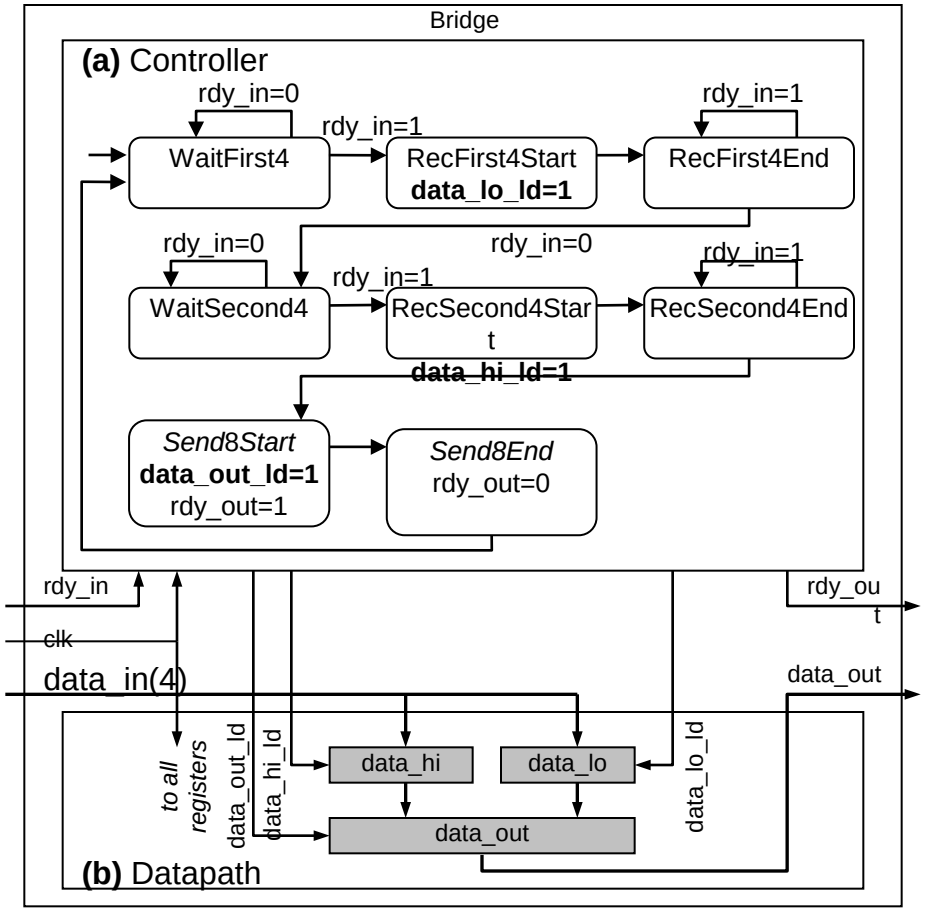
a view inside the controller and datapath

RT-level custom single-purpose processor design

- We often start with a state machine
  - Rather than algorithm
  - Cycle timing often too central to functionality
- Example
  - Bus bridge that converts 4-bit bus to 8-bit bus
  - Start with FSMD
  - Known as register-transfer (RT) level
  - Exercise: complete the design



RT-level custom single-purpose processor design (cont')



Optimizing single-purpose processors

- Optimization is the task of making design metric values the best possible
- Optimization opportunities
  - original program
  - FSMD
  - datapath
  - FSM

Optimizing the original program

- Analyze program attributes and look for areas of possible improvement
  - number of computations
  - size of variable
  - time and space complexity
  - operations used
    - multiplication and division very expensive

Optimizing the original program (cont')

original program

```
 0: int  x, y;
 1: while (1) {
 2:   while (!go_i);
 3:   x = x_i;
 4:   y = y_i;
 5:   while  (x != y)  {
 6:       if  (x < y)
 7:           y = y - x;
          else
 8:           x = x - y;
      }
 9:   d_o = x;
   }
```

replace the subtraction operation(s) with modulo operation in order to speed up program

optimized program

```
 0: int  x, y, r;
 1: while (1) {
 2:   while (!go_i);
      // x must be the larger
number
 3:   if (x_i >= y_i)  {
 4:      x=x_i;
 5:      y=y_i;
      }
 6:   else {
 7:      x=y_i;
 8:      y=x_i;
      }
 9:   while  (y != 0)  {
10:      r = x % y;
11:      x = y;
12:      y = r;
      }
13:   d_o = x;
   }
```

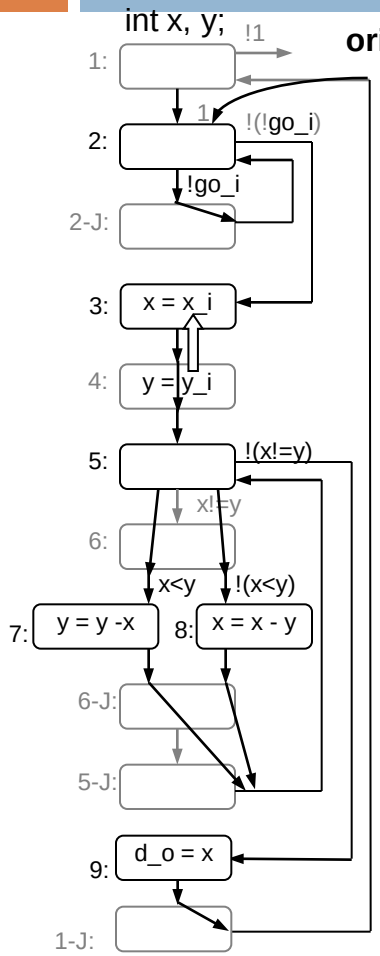
GCD(42, 8) - 9 iterations to complete the loop
x and y values evaluated as follows : (42, 8), (43, 8), (26,8), (18,8), (10, 8), (2,8), (2,6), (2,4), (2,2).

GCD(42,8) - 3 iterations to complete the loop
x and y values evaluated as follows: (42, 8), (8,2), (2,0)

Optimizing the FSMD

- Areas of possible improvements
  - merge states
    - states with constants on transitions can be eliminated, transition taken is already known
    - states with independent operations can be merged
  - separate states
    - states which require complex operations (a*b*c*d) can be broken into smaller states to reduce hardware size
  - scheduling

Optimizing the FSMD (cont.)



original FSMD

eliminate state 1 – transitions have constant values

merge state 2 and state 2J – no loop operation in between them

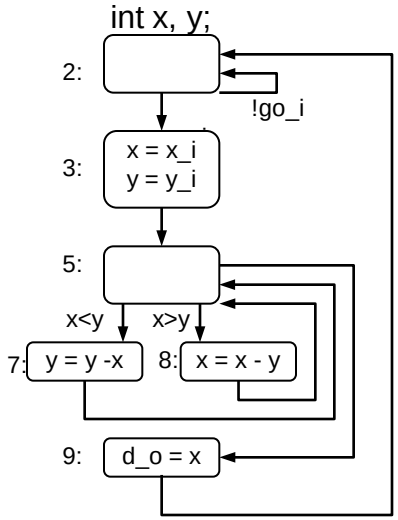
merge state 3 and state 4 – assignment operations are independent of one another

merge state 5 and state 6 – transitions from state 6 can be done in state 5

eliminate state 5J and 6J – transitions from each state can be done from state 7 and state 8, respectively

eliminate state 1-J – transition from state 1-J can be done directly from state 9

optimized FSMD



Optimizing the datapath

- Sharing of functional units
  - one-to-one mapping, as done previously, is not necessary
  - if same operation occurs in different states, they can share a single functional unit
- Multi-functional units
  - ALUs support a variety of operations, it can be shared among operations occurring in different states

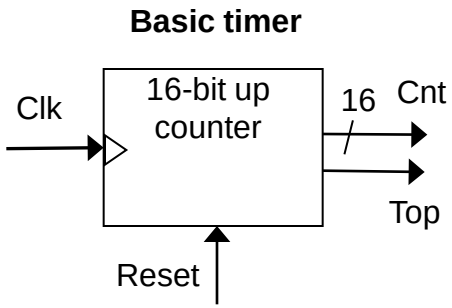
Optimizing the FSM

- State encoding
  - task of assigning a unique bit pattern to each state in an FSM
  - size of state register and combinational logic vary
  - can be treated as an ordering problem
- State minimization
  - task of merging equivalent states into a single state
    - state equivalent if for all possible input combinations the two states generate the same outputs and transitions to the next same state

Peripherials

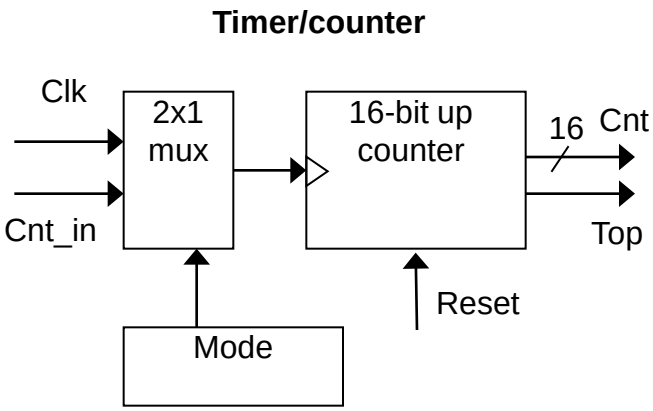
Timers, counters, watchdog timers

- Timer: measures time intervals
  - To generate timed output events
    - e.g., hold traffic light green for 10 s
  - To measure input events
    - e.g., measure a car's speed
- Based on counting clock pulses
  - E.g., let Clk period be 10 ns
  - And we count 20,000 Clk pulses
  - Then 200 microseconds have passed
  - 16-bit counter would count up to 65,535*10 ns = 655.35 microsec., resolution = 10 ns
  - Top: indicates top count reached, wrap-around



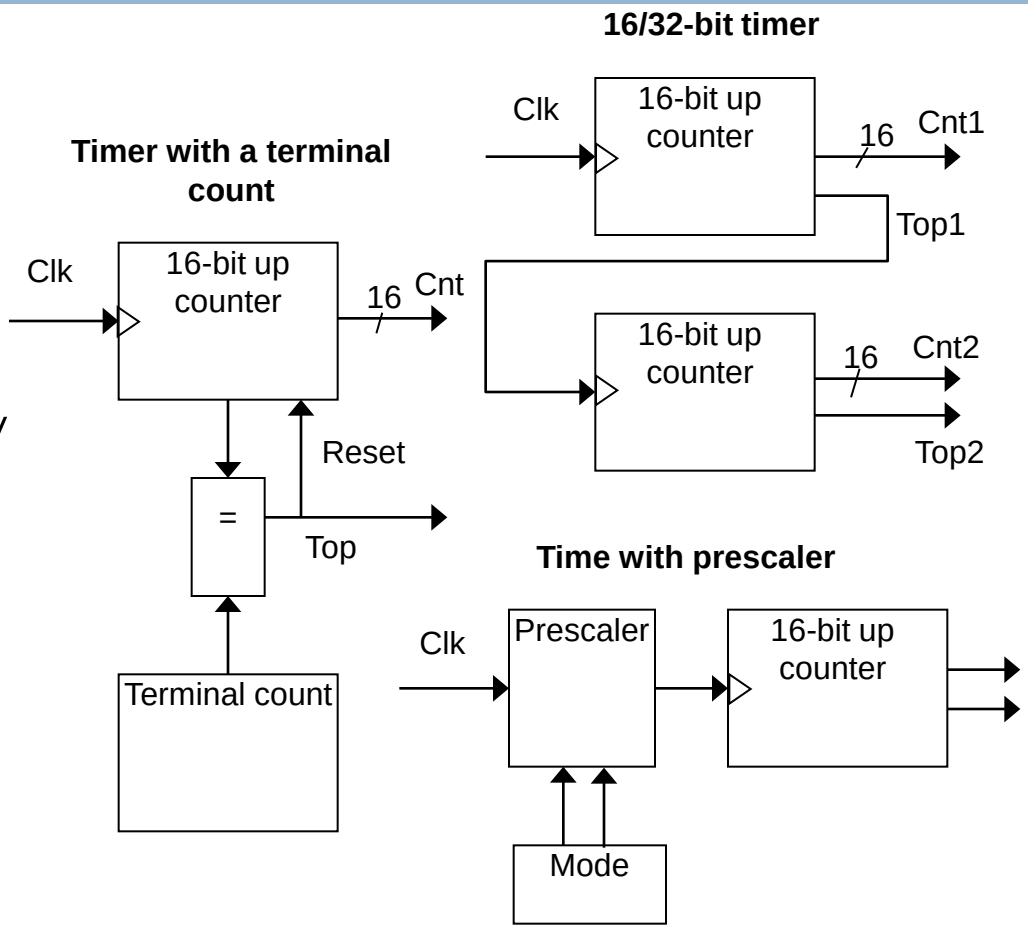
Counters

- Counter: like a timer, but counts pulses on a general input signal rather than clock
  - e.g., count cars passing over a sensor
  - Can often configure device as either a timer or counter

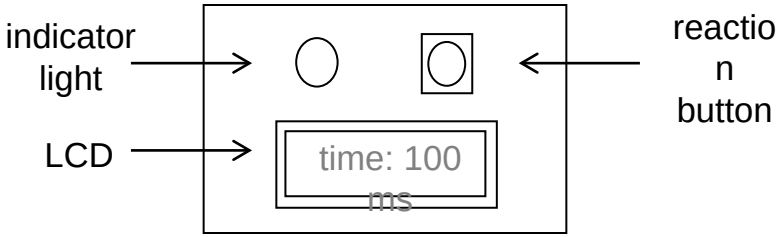


Other timer structures

- Interval timer
  - Indicates when desired time interval has passed
  - We set terminal count to desired interval
    - *Number of clock cycles = Desired time interval / Clock period*
- Cascaded counters
- Prescaler
  - Divides clock
  - Increases range, decreases resolution



Example: Reaction Timer

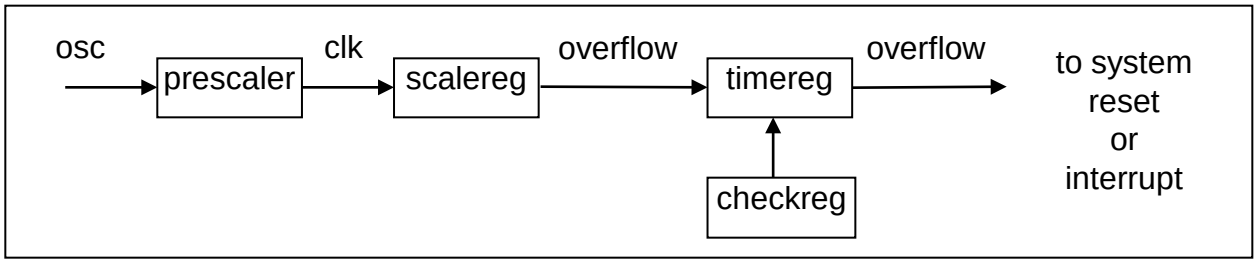


- Measure time between turning light on and user pushing button
  - 16-bit timer, clk period is 83.33 ns, counter increments every 6 cycles
  - Resolution = 6*83.33=0.5 microsec.
  - Range = 65535*0.5 microseconds = 32.77 milliseconds
  - Want program to count millisec., so initialize counter to 65535 – 1000/0.5 = 63535

```
/* main.c */

#define MS_INIT       63535
void main(void){
   int count_milliseconds = 0;

   configure timer mode
   set Cnt to MS_INIT

   wait a random amount of time
   turn on indicator light
   start timer

while (user has not pushed reaction button){
   if(Top) {
      stop timer
      set Cnt to MS_INIT
      start timer
      reset Top
      count_milliseconds++;
   }
 }
 turn light off
 printf("time:  %i ms", count_milliseconds);
}
```

Watchdog timer

- Must reset timer every X time unit, else timer generates a signal
- Common use: detect failure, self-reset
- Another use: timeouts
  - e.g., ATM machine
  - 16-bit timer, 2 microsec. resolution
  - *timereg* value = $2*(2^{16}-1)-X = 131070-X$
  - For 2 min., X = 120,000 microsec.

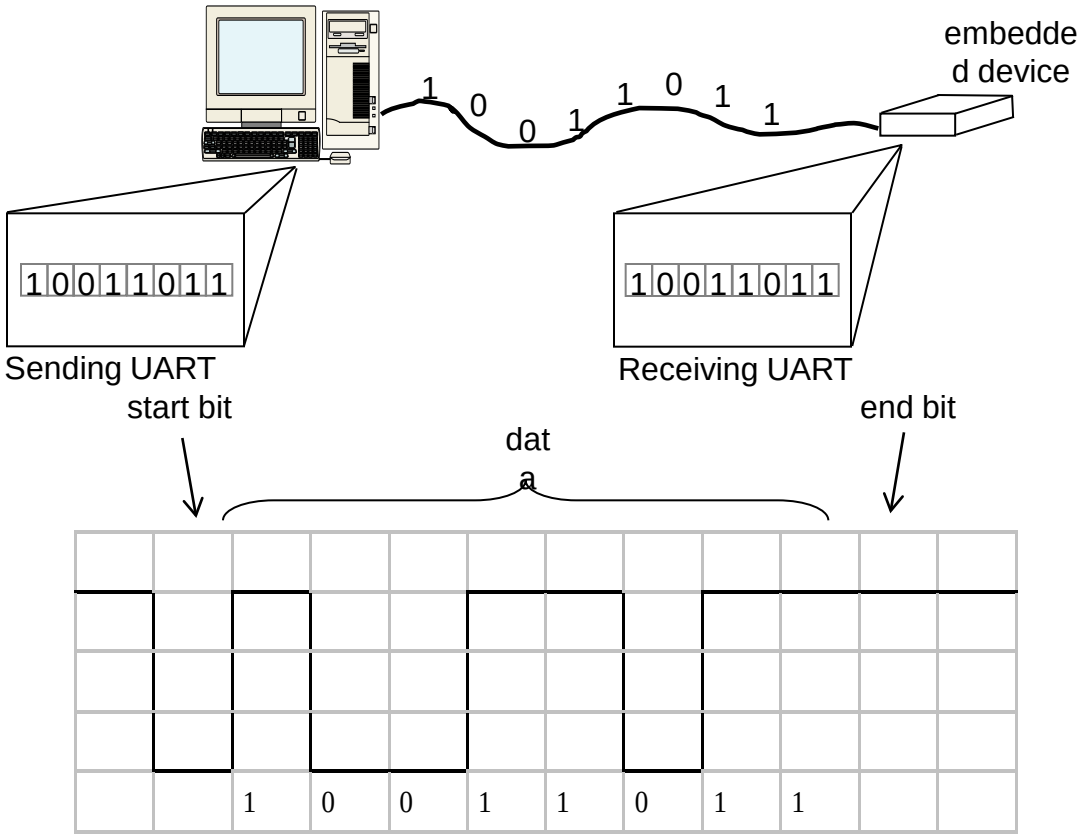



```
/* main.c */

main(){
  wait until card inserted
  call watchdog_reset_routine

  while(transaction in progress){
    if(button pressed){
      perform corresponding action
      call watchdog_reset_routine
    }

/* if watchdog_reset_routine not called
every < 2 minutes,
interrupt_service_routine is called */
}
```

```
watchdog_reset_routine(){
/* checkreg is set so we can load value
into timereg.  Zero is loaded into
scalereg and 11070 is loaded into
timereg */

  checkreg = 1
  scalereg = 0
  timereg  = 11070
}

void interrupt_service_routine(){
  eject card
  reset screen
}
```

Serial Transmission Using UARTs

- UART: Universal Asynchronous Receiver Transmitter
  - Takes parallel data and transmits serially
  - Receives serial data and converts to parallel
- Parity: extra bit for simple error checking
- Start bit, stop bit
- Baud rate
  - signal changes per second
  - bit rate usually higher



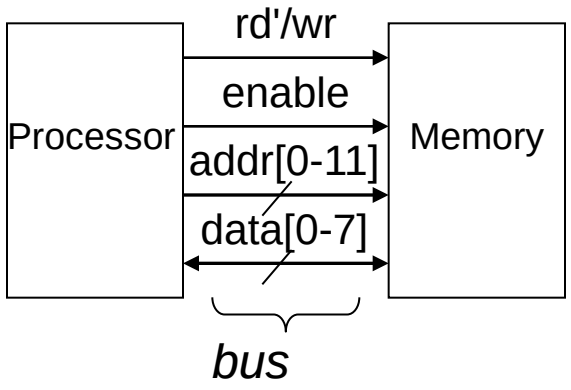
Interfacing

Bus Overview

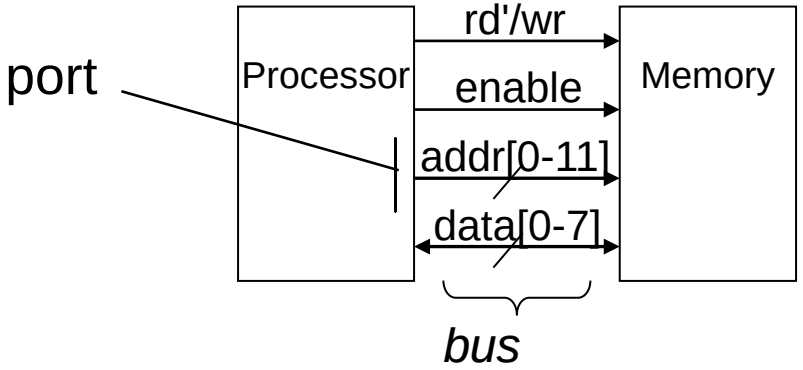
AHB Bus

A simple bus

- Wires:
  - Uni-directional or bi-directional
  - One line may represent multiple wires
- Bus
  - Set of wires with a single function
    - Address bus, data bus
  - Or, entire collection of wires
    - Address, data and control
    - Associated protocol: rules for communication



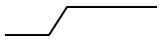
Ports



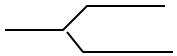
- Conducting device on periphery
- Connects bus to processor or memory
- Often referred to as a *pin*
  - Actual pins on periphery of IC package that plug into socket on printed-circuit board
  - Sometimes metallic balls instead of pins
  - Today, metal "pads" connecting processors and memories within single IC
- Single wire or set of wires with single function
  - E.g., 12-wire address port

Timing Diagrams

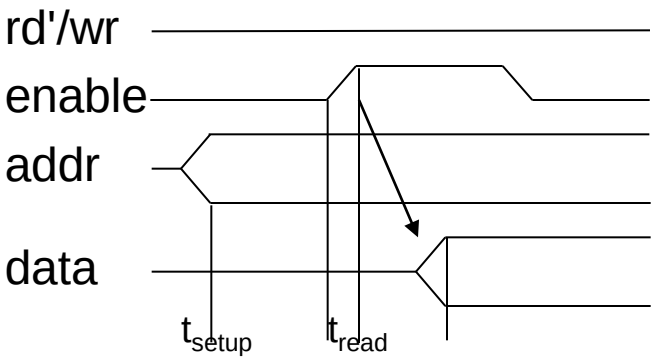
- Most common method for describing a communication protocol
- Time proceeds to the right on x-axis
- Control signal: low or high



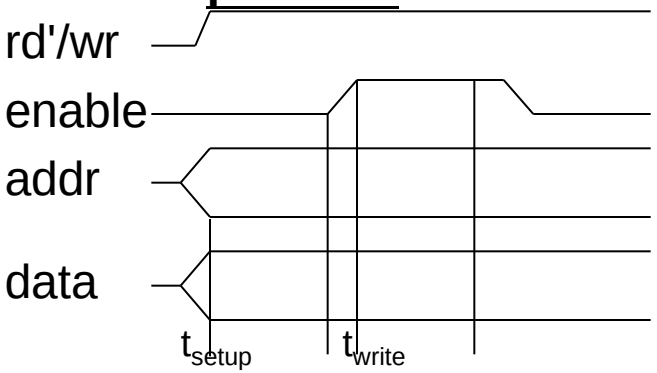
  - May be active low (e.g., go', /go, or go_L)
  - Use terms *assert* (active) and *deassert*
  - Asserting go' means go=0



- Data signal: not valid or valid
- Protocol may have subprotocols
  - Called bus cycle, e.g., read and write
  - Each may be several clock cycles
- Read example
  - *rd'/wr* set low,address placed on *addr* for at least t_{setup} time before *enable* asserted, enable triggers memory to place data on *data* wires by time t_{read}



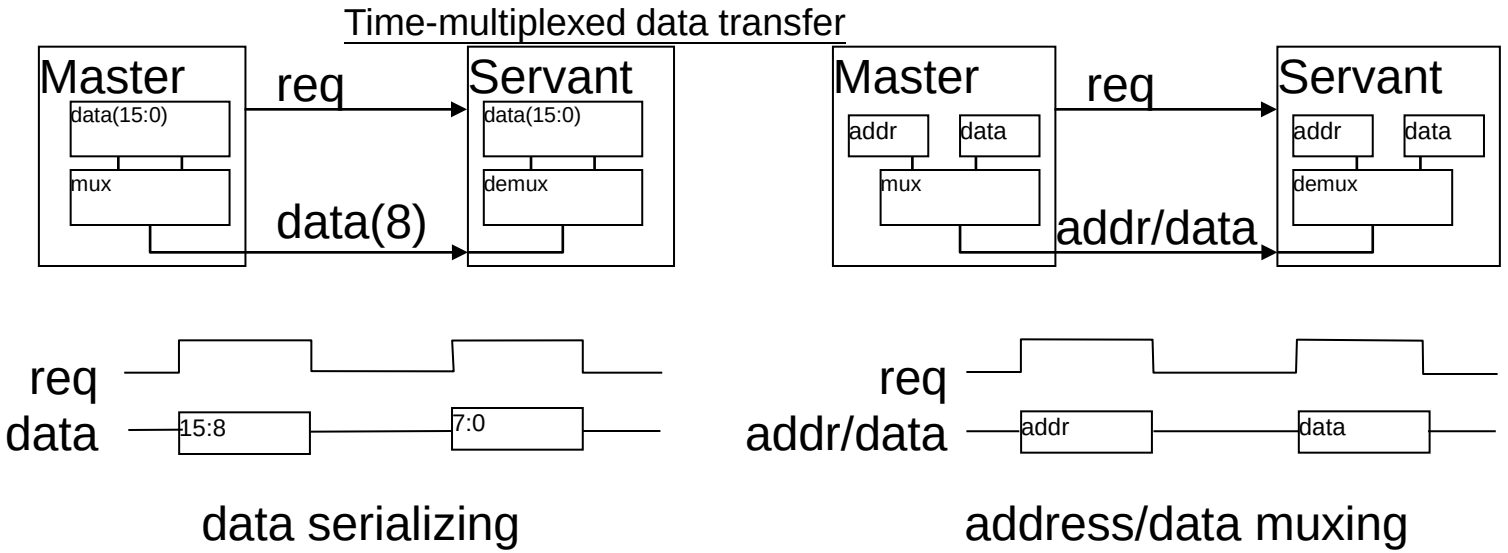
read protocol



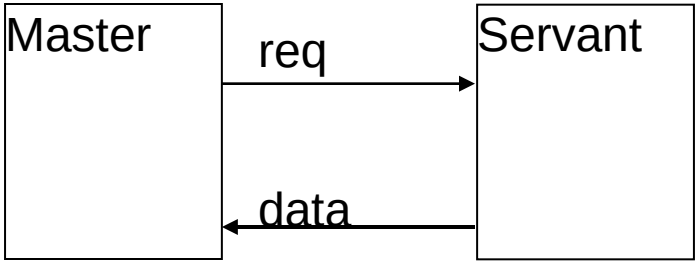
write protocol

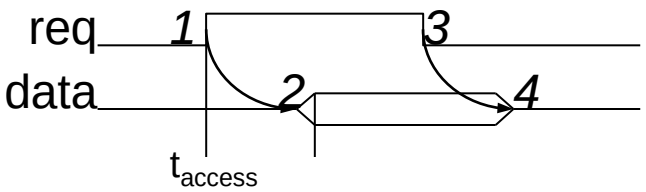
Basic protocol concepts

- Actor: master initiates, servant (slave) respond
- Direction: sender, receiver
- Addresses: special kind of data
  - Specifies a location in memory, a peripheral, or a register within a peripheral
- Time multiplexing
  - Share a single set of wires for multiple pieces of data
  - Saves wires at expense of time



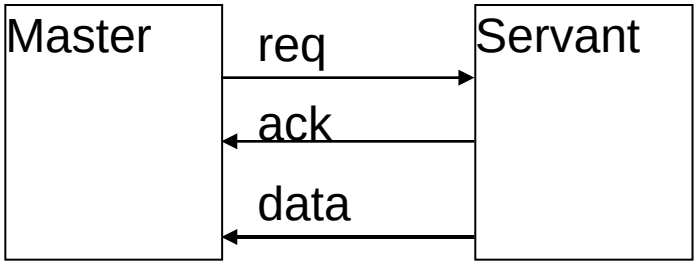
Basic protocol concepts: control methods

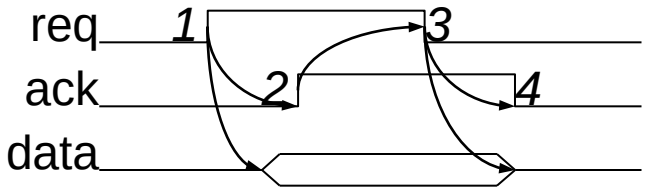




1. Master asserts *req* to receive data
2. Servant puts data on bus **within time t_{access}**
3. Master receives data and deasserts *req*
4. Servant ready for next request

Strobe protocol

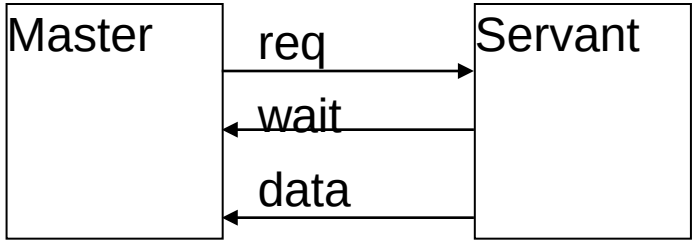


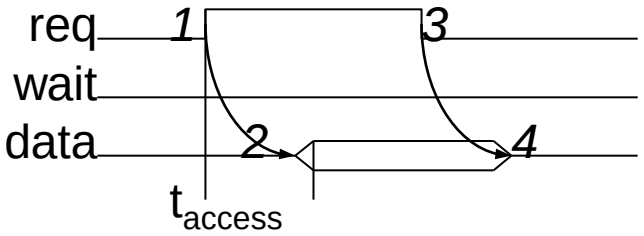


1. Master asserts *req* to receive data
2. Servant puts data on bus **and asserts *ack***
3. Master receives data and deasserts *req*
4. Servant ready for next request

Handshake protocol

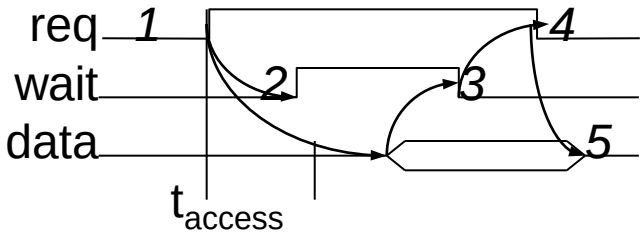
A strobe/handshake compromise





1. Master asserts *req* to receive data
2. Servant puts data on bus **within time t_{access}** (wait line is unused)
3. Master receives data and deasserts *req*
4. Servant ready for next request

Fast-response case



1. Master asserts *req* to receive data
2. Servant can't put data within **t_{access}**, **asserts *wait*** ack
3. Servant puts data on bus and **deasserts *wait***
4. Master receives data and deasserts *req*
5. Servant ready for next request

Slow-response case

AHB Master Behavioral Model

Bus protocol

AMBA

AHB characteristics and infrastructure

Control signals

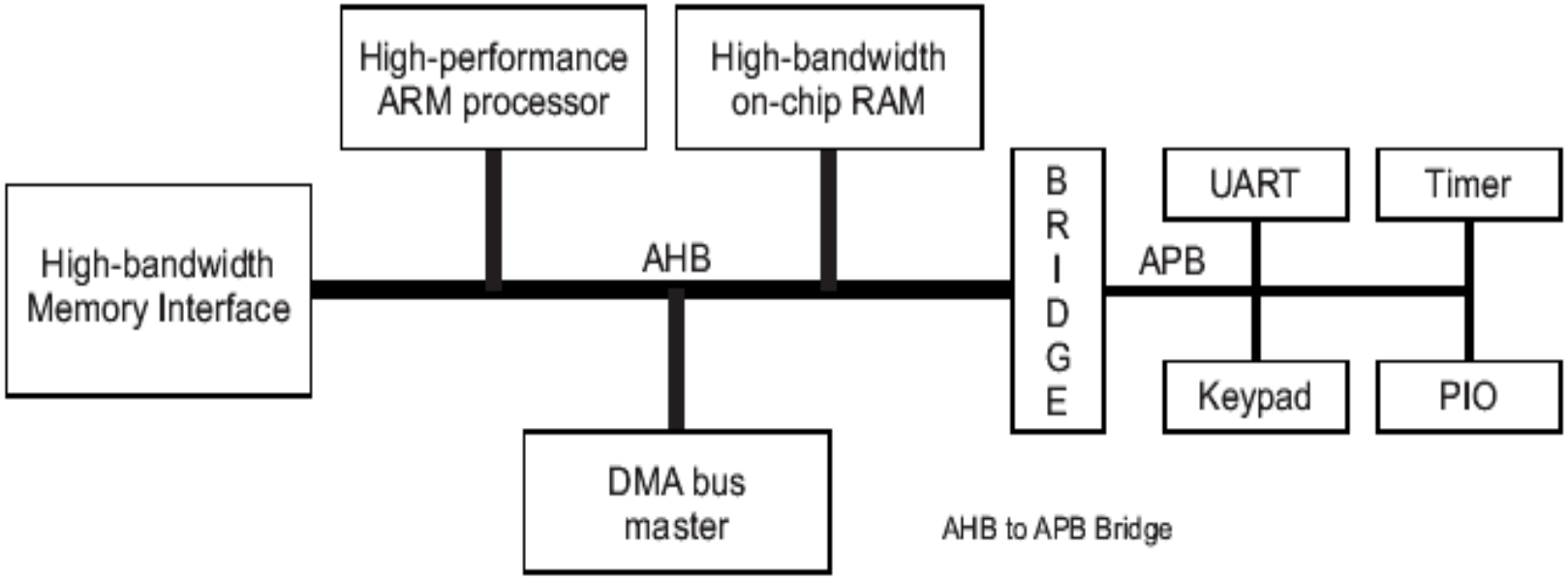
Ref:
-- AMBA 2.0
-- 簡弘倫, “Verilog 晶片設計,” 文魁資訊, 2005

Bus Protocols

- Specification of signals, timing, and sequencing of bus operations
  - Allows independent design of components
  - Ensures interoperability
- Standard bus protocols
  - PCI, VXI, …
    - For connecting boards in a system
  - AMBA (ARM), CoreConnect (IBM), Wishbone (Open Cores)
    - For connecting blocks within a chip

AMBA

- Advanced High-performance Bus(AHB)
- Advanced System Bus(ASB)
- Advanced Peripheral Bus



AHB characteristic

- Single cycle edge operation
- Non-tristate implementation
- Burst transfers
- Split transactions
- Single cycle bus master handover
- Wider data bus configurations(64/128bit)

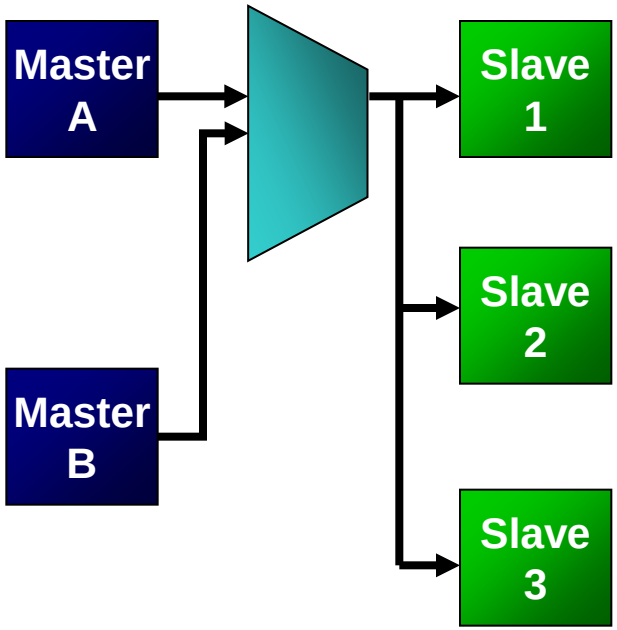
AHB Simple framework

address
Control signal
Write data

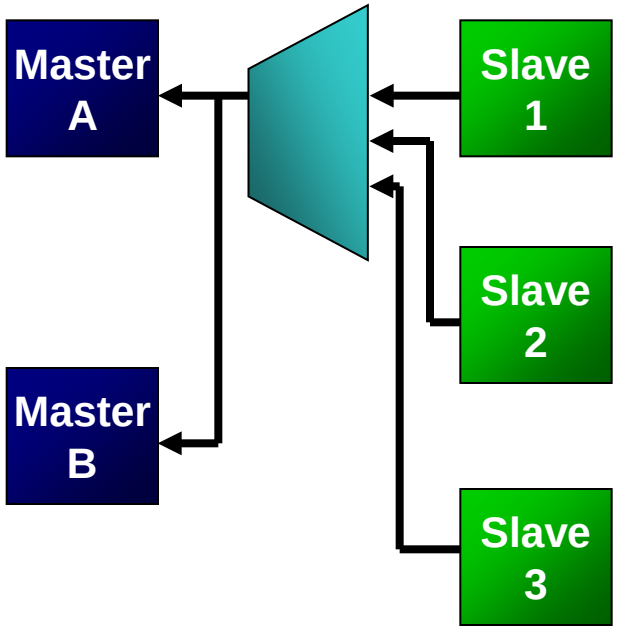
clock

arbitration

Read data
Response signal

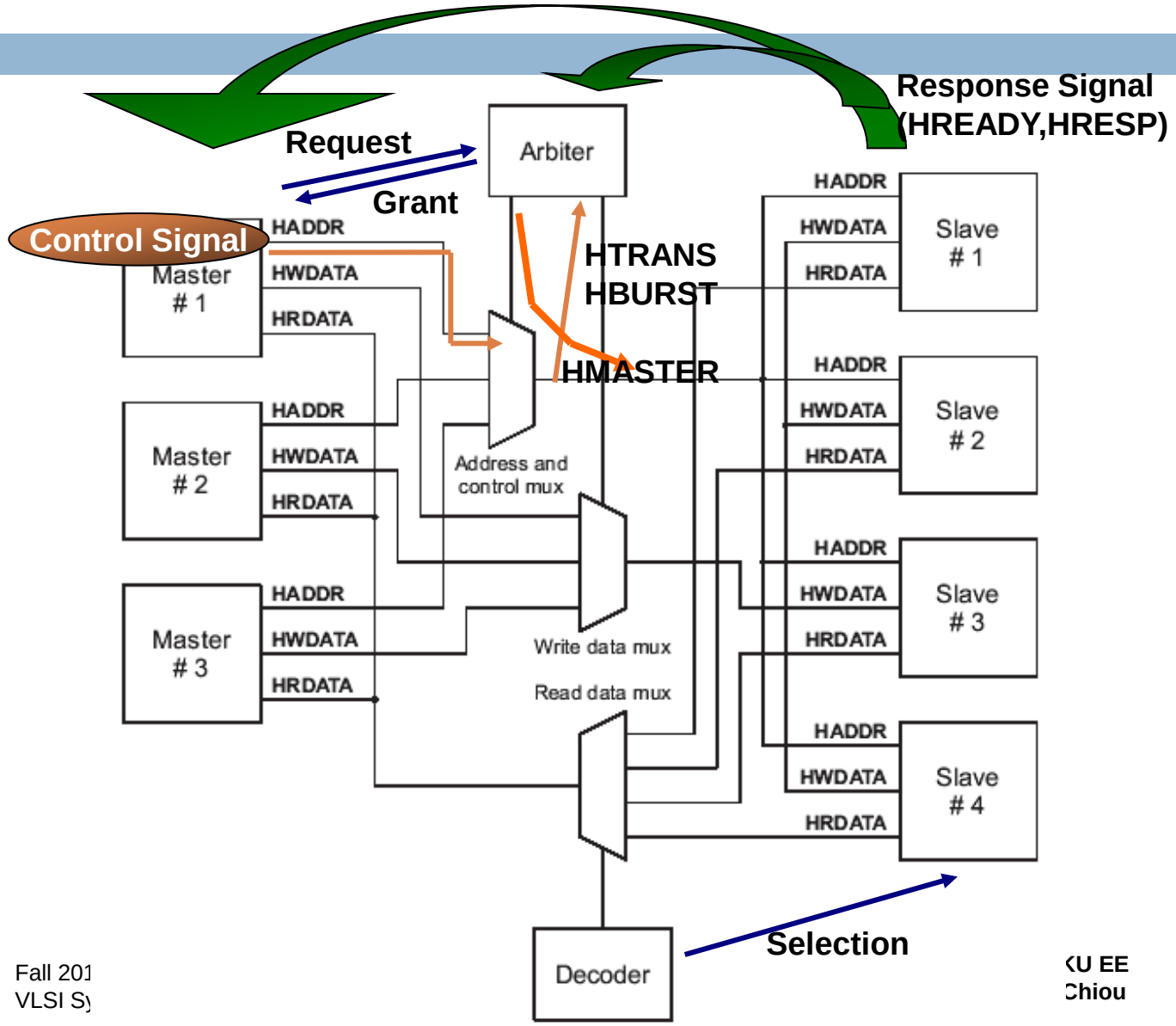


Master to Slave Multiplexor

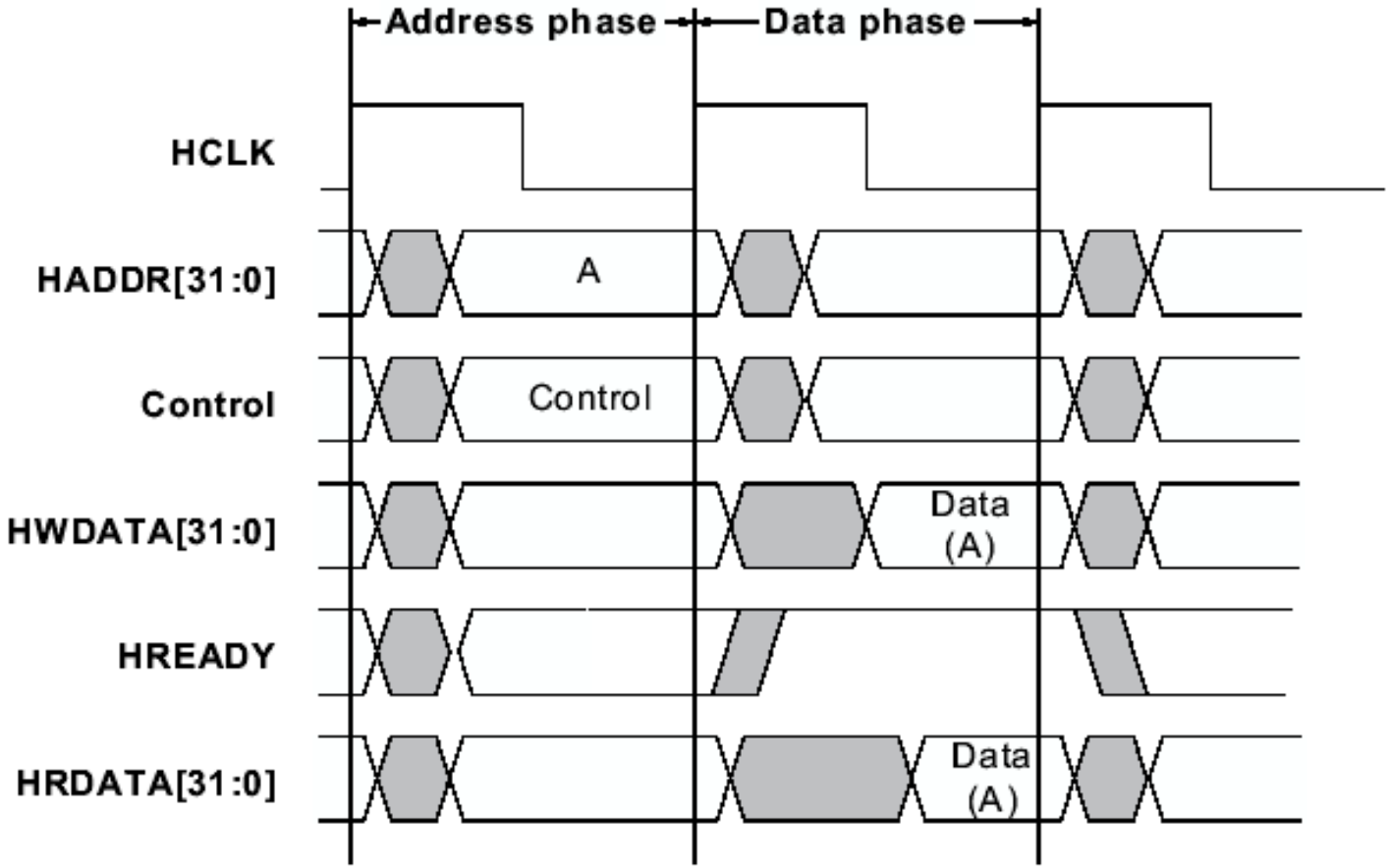


Slave to Master Multiplexor

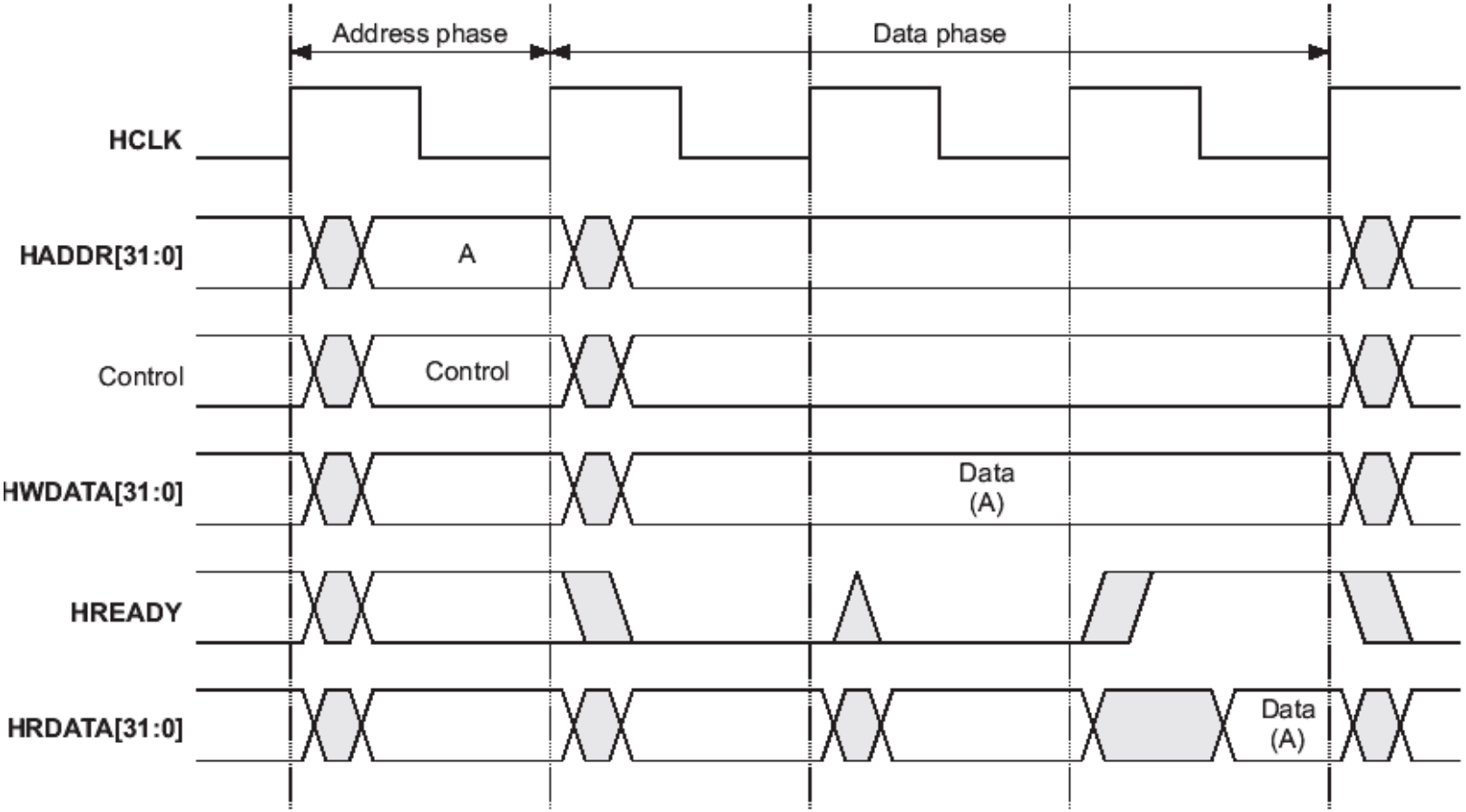
AHB Bus Interconnection



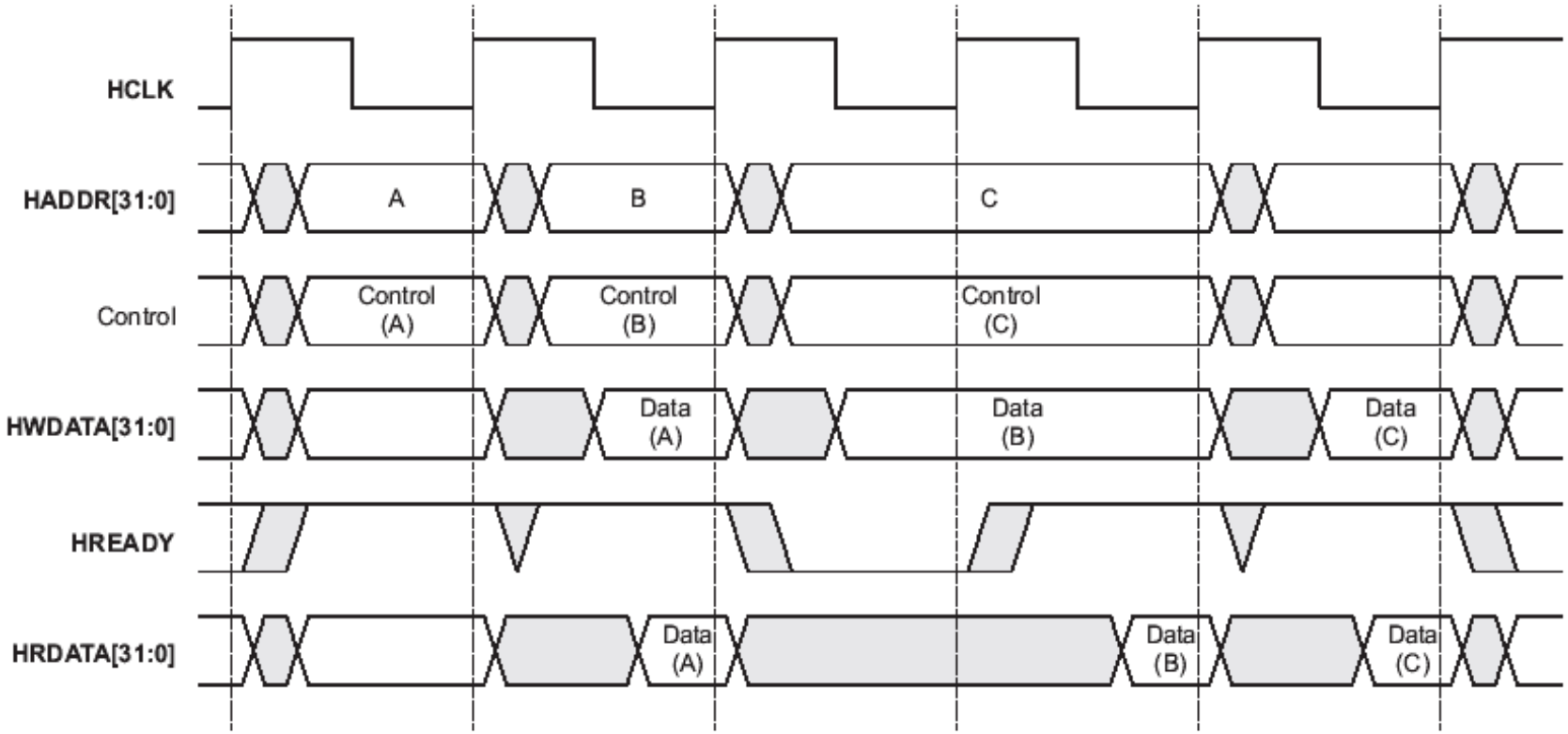
Basic Transfer (no wait state)



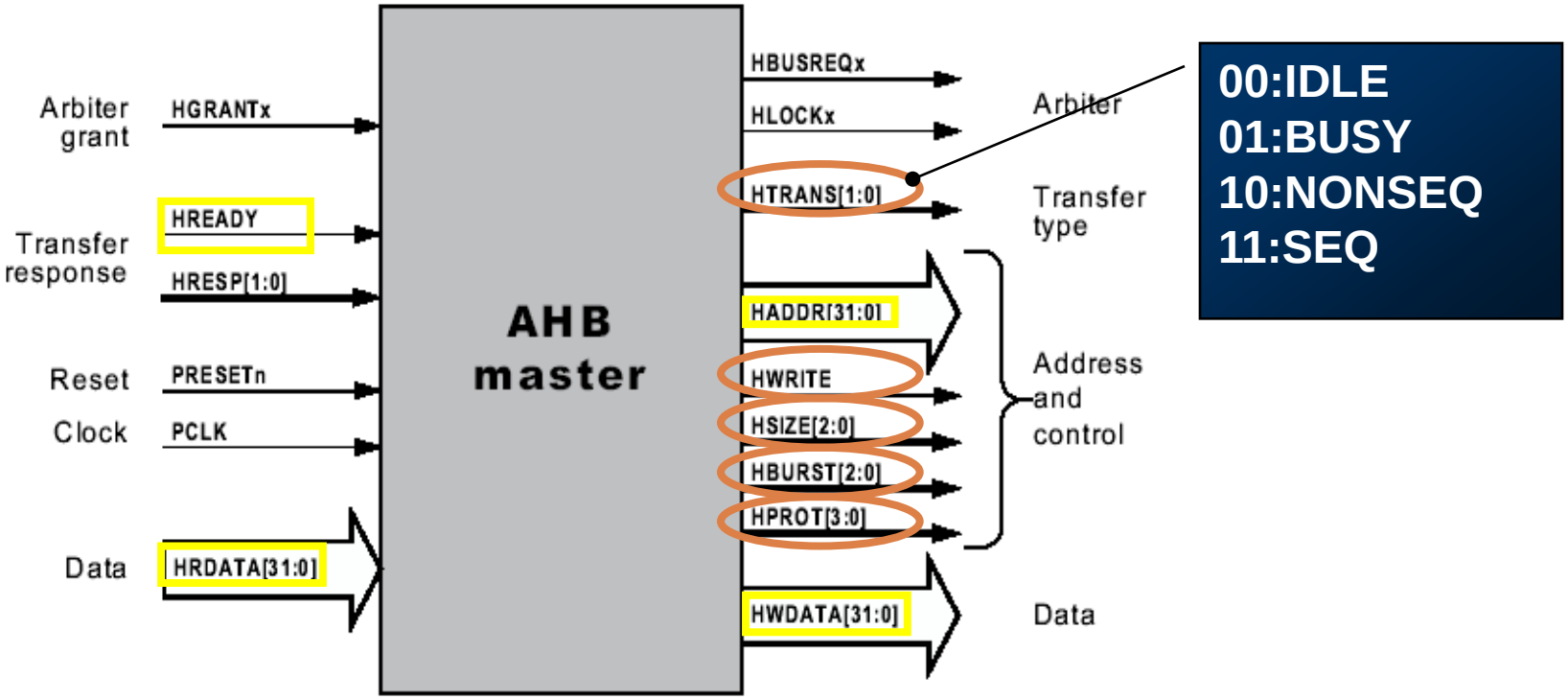
Basic Transfer (wait state)



Multiple Transfer

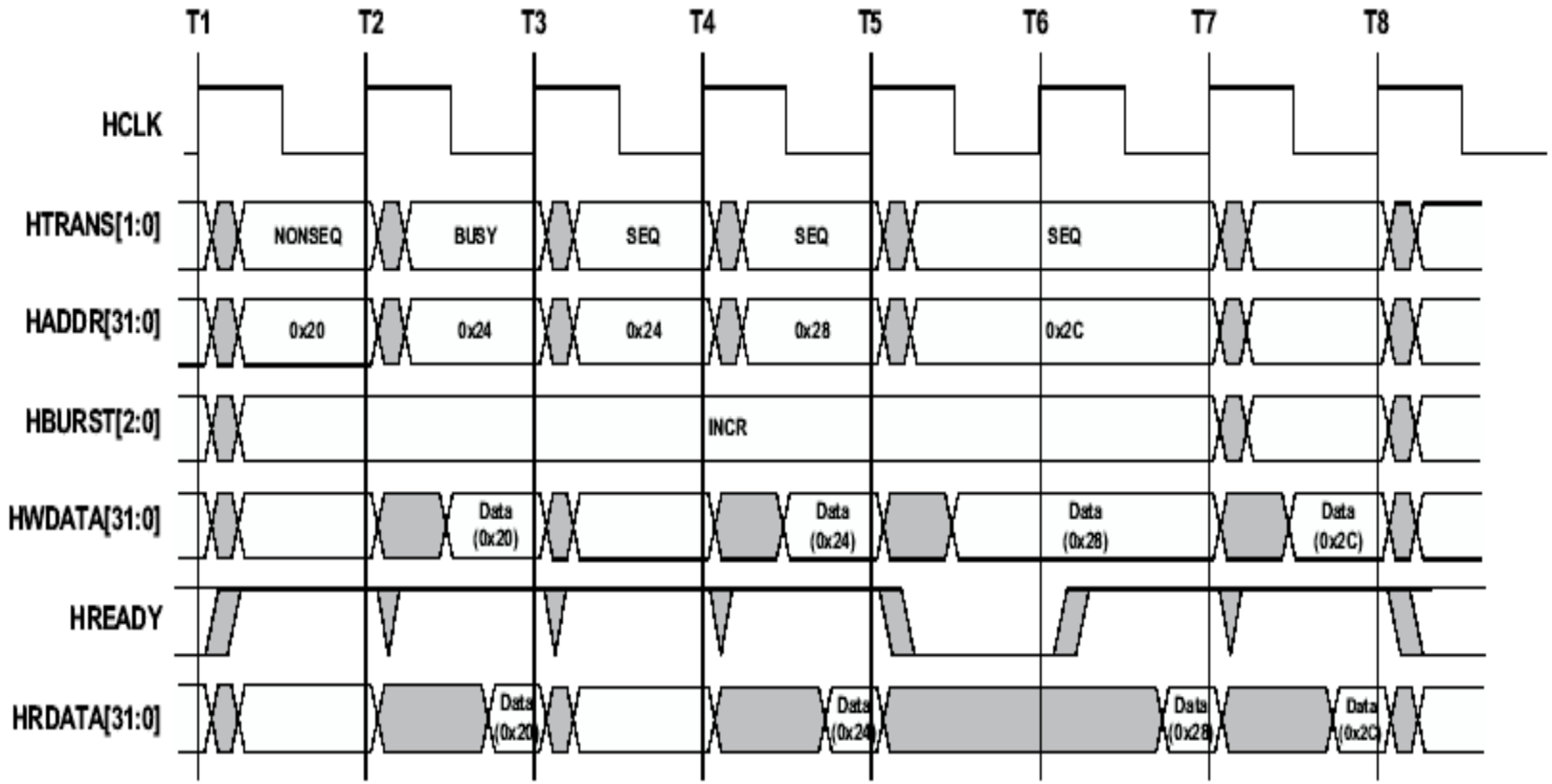


Master---Transfer Type



IDLE : Master has no data to transfer,而Slave會在data phase回應OKAY(response signal)
BUSY: Master無法準備好資料在下一個週期傳送,則Master發出BUSY訊號來延遲這筆資料的傳送,而Slave會在data phase回應OKAY(response signal)
NONSEQ : 表目前transfer的address/control signal和前週期被傳送的資料無關
SEQ :表目前transfer的address/control signal和前週期被傳送的資料相關) (用於Burst transfer)

Transfer type example

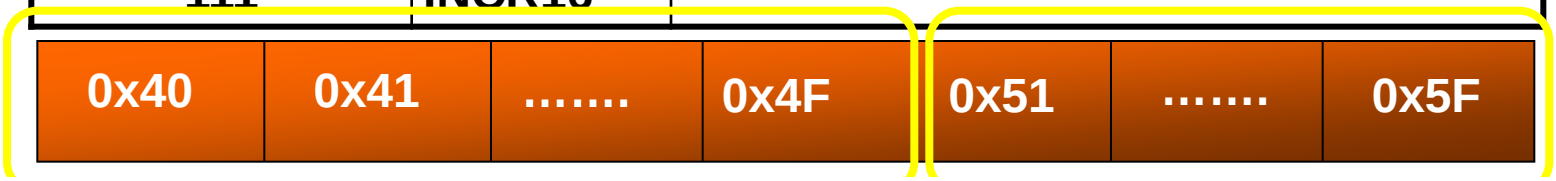


HSIZE Operation

HSIZE[2:0]	SIZE (bit)
000	8
001	16
010	32
011	64
100	128
101	256
110	512
111	1024

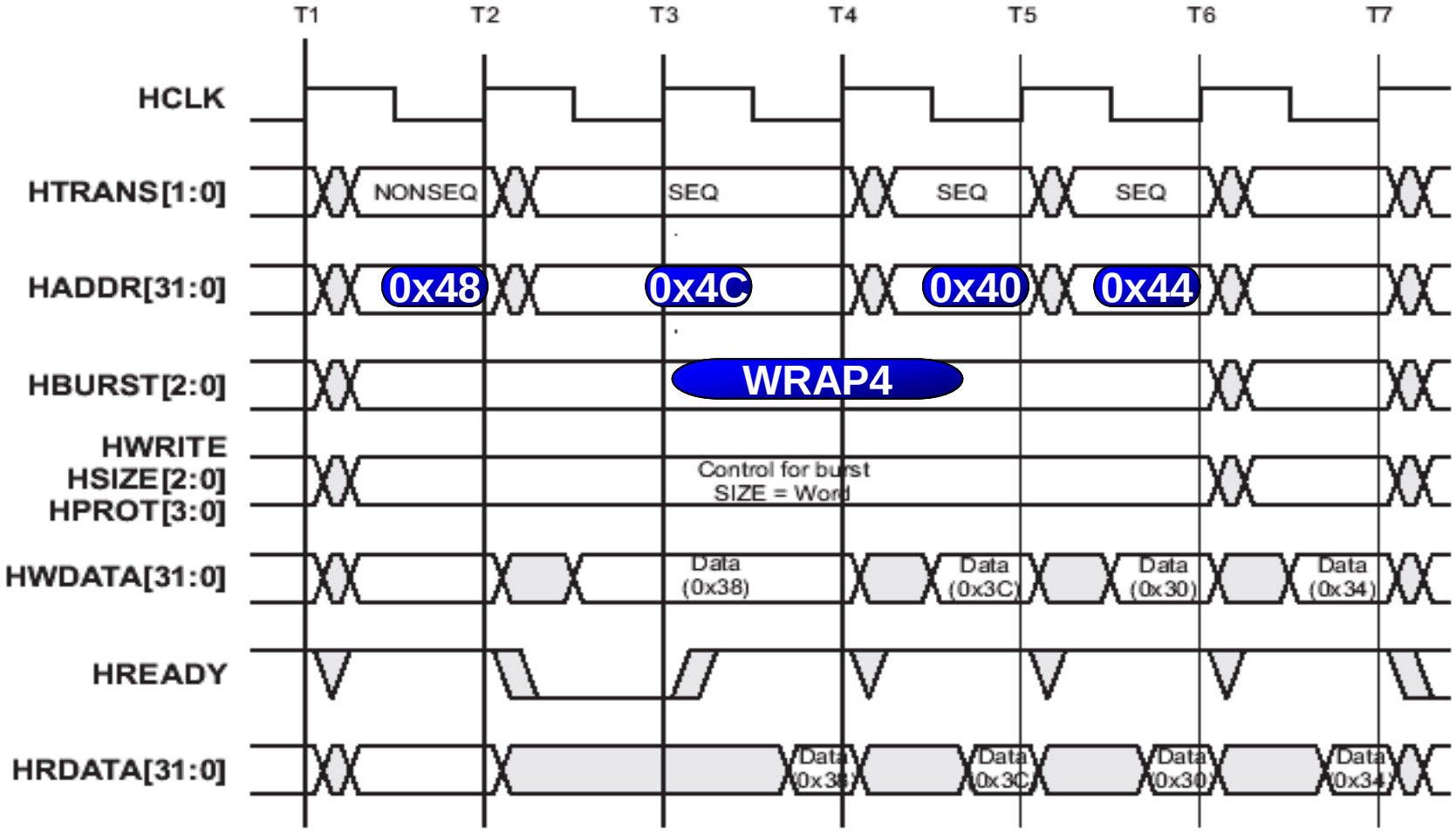
Burst Operation

HBURST[2:0]	Type	Sample(HSIZE=4byte)
000	SINGLE	0x48
001	INCR	0x48,0x4c,0x50
010	WRAP4	0x48, 0x4c,0x40,0x44
011	INCR4	0x48, 0x4c,0x50,0x54
100	WRAP8	
101	INCR8	
110	WRAP16	
111	INCR16	

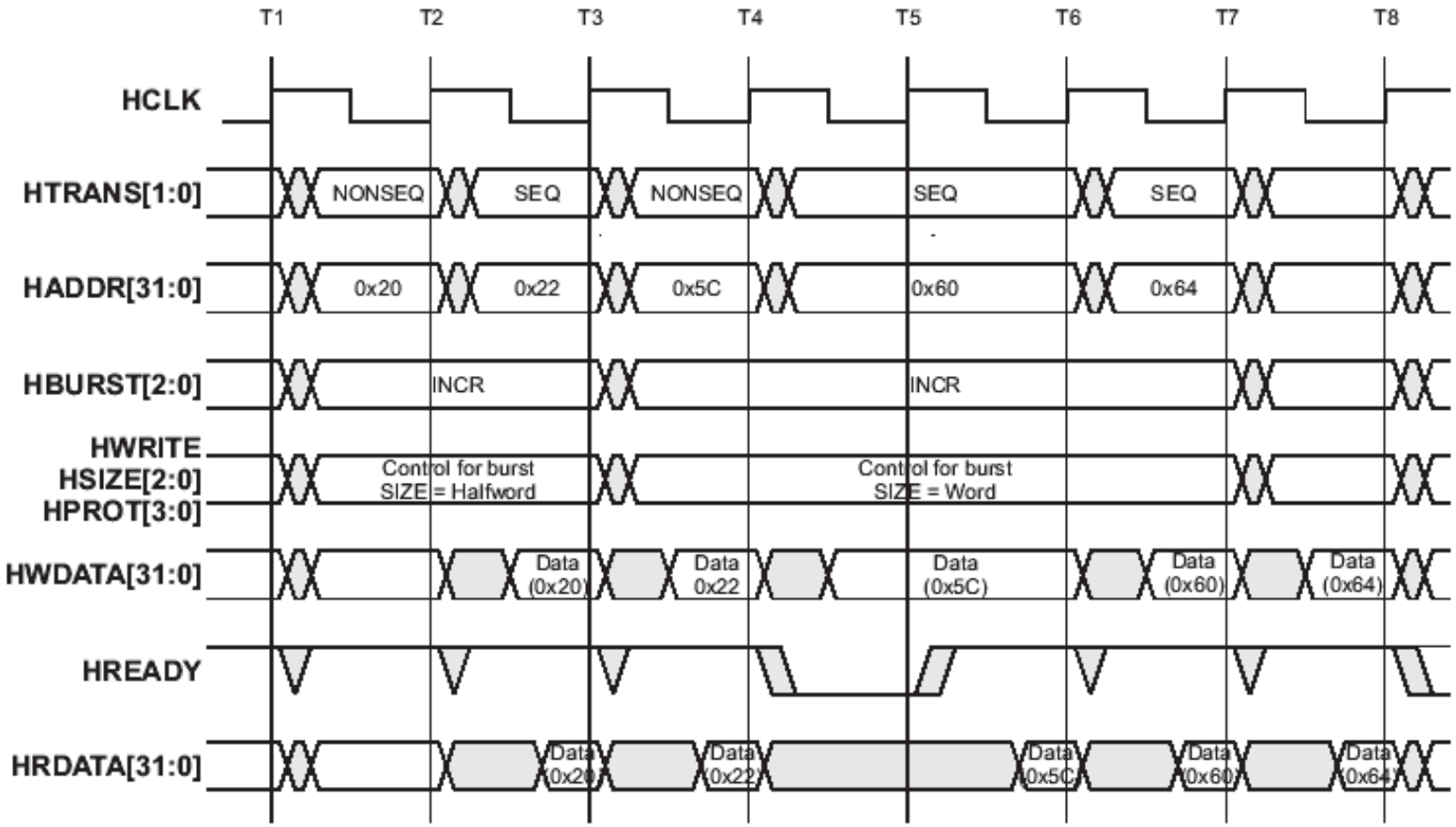


Increase (INCR) : 將前一筆的資料位址加上HSIZE的大小即為下一筆資料的位址
WRAP : wrapping burst將memory切割成某固定大小的一個個memory boundary,當transfer address要跨越此boundary時,下一筆transfer address會繞回原之boundary的起點

Burst Operation example

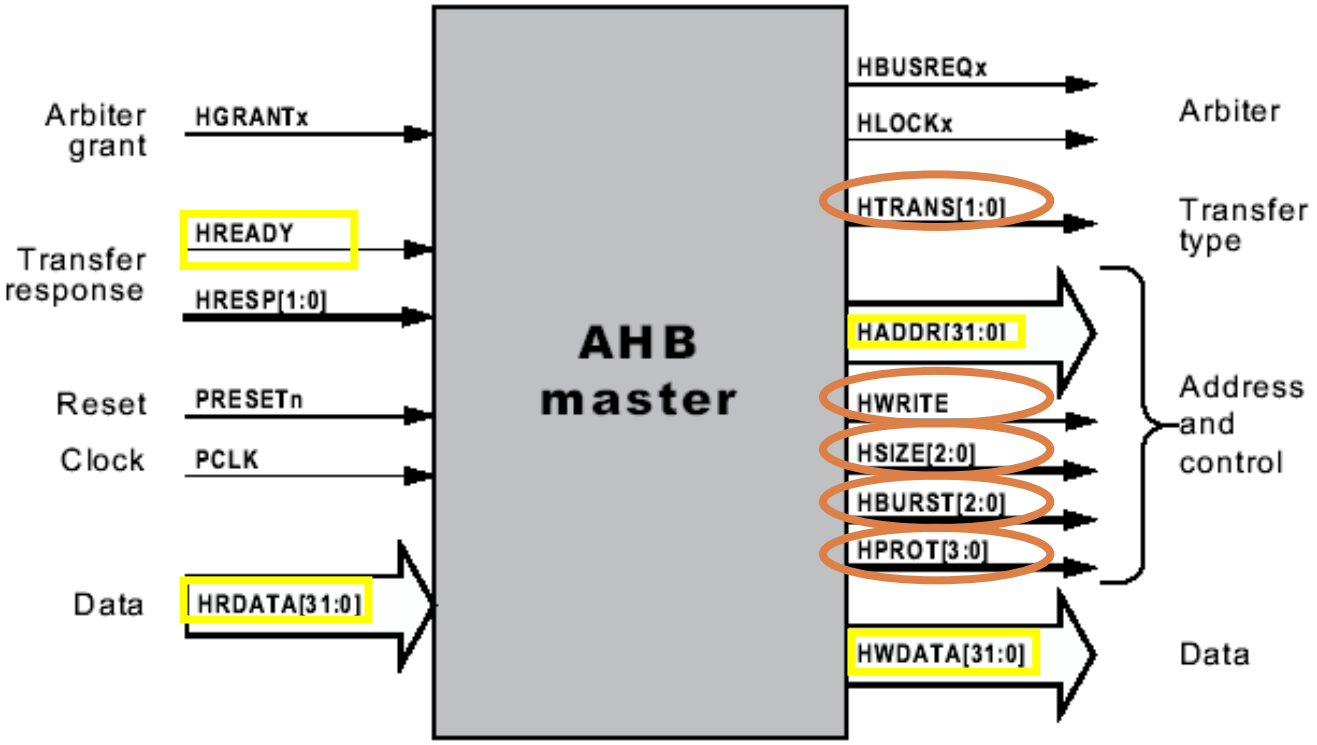


Burst Operation example



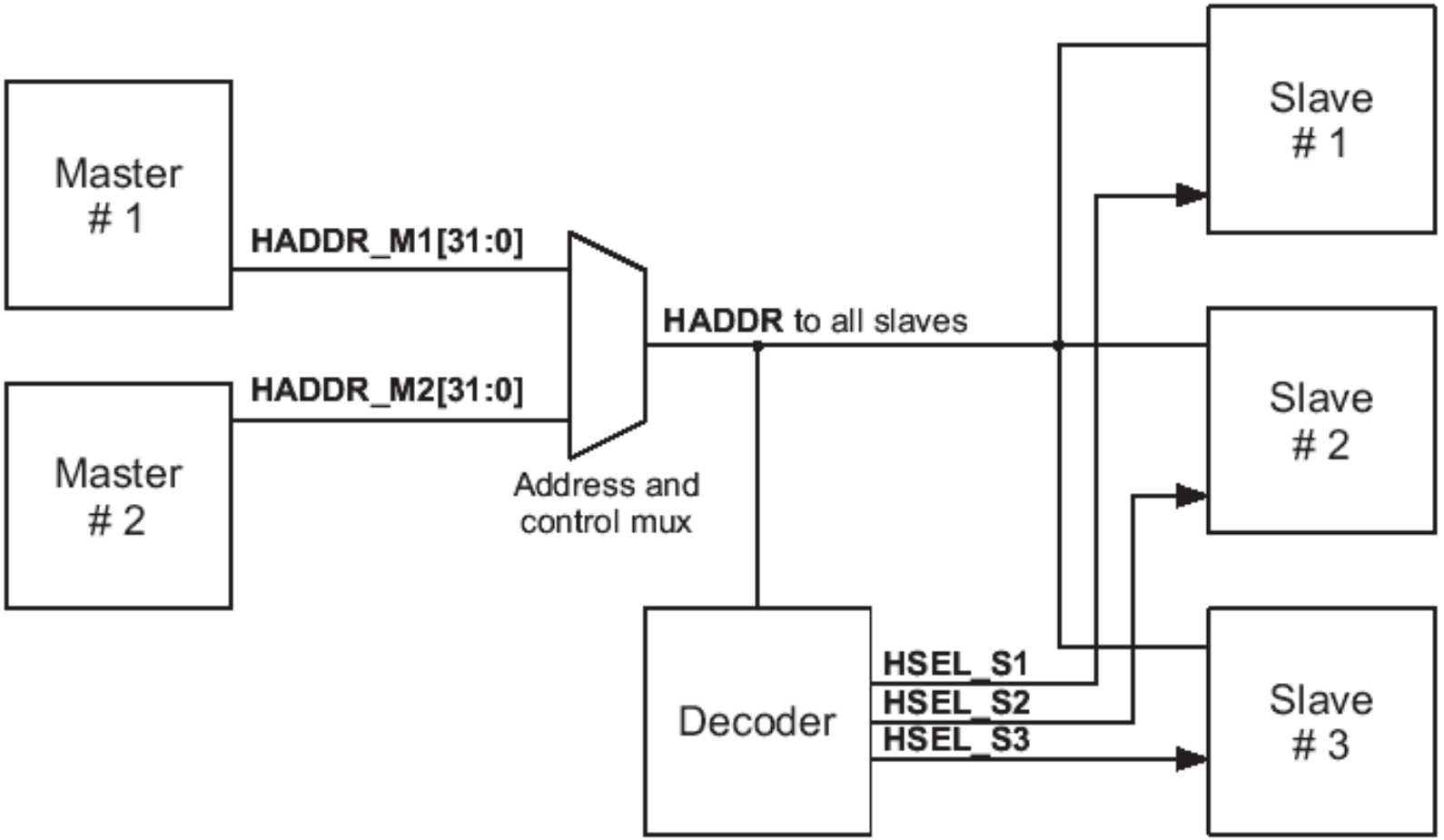
Undefined-length burst

Master---Others

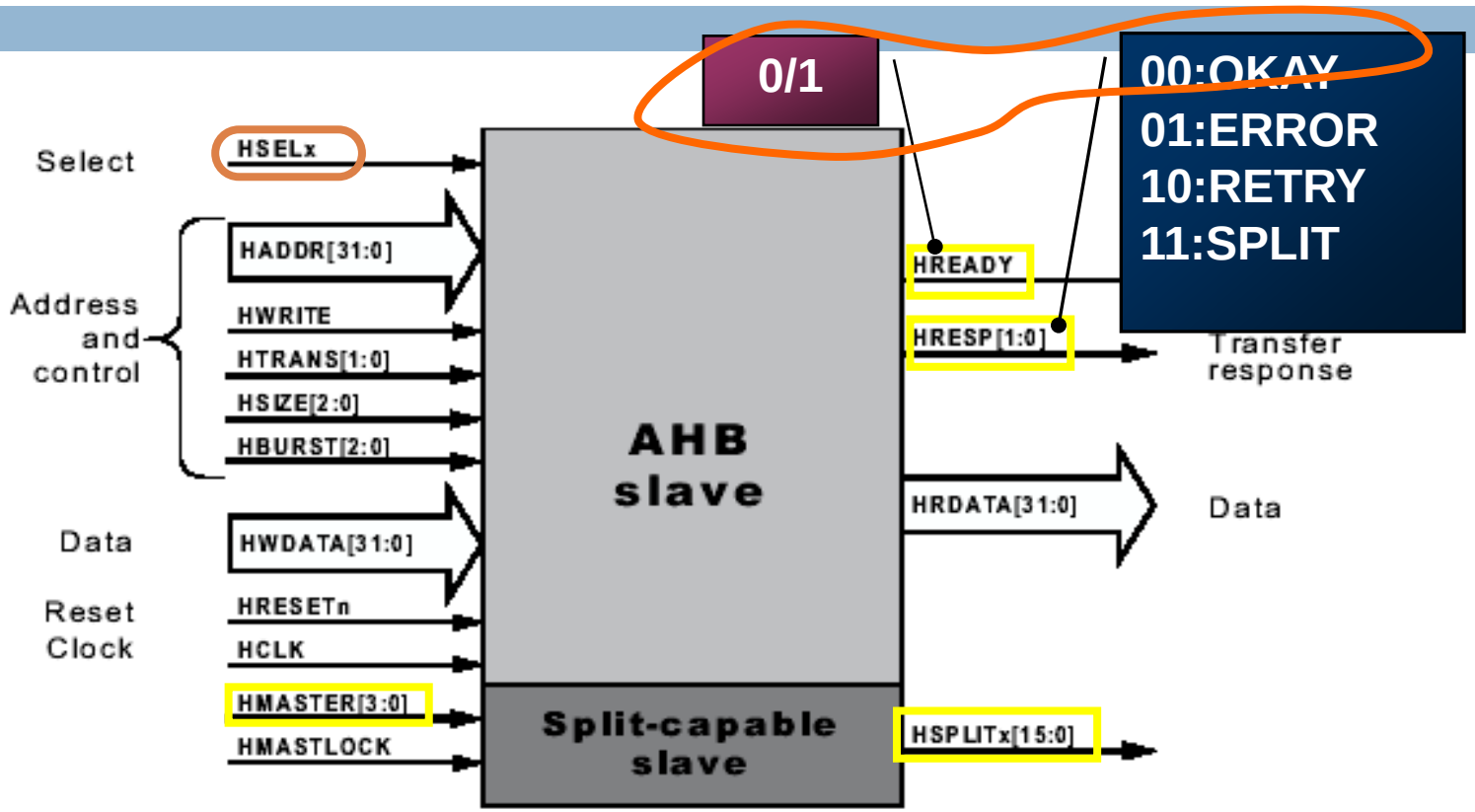


HBUSREQx : Master傳給Arbiter的編號
HWRITE : Write/Read (1/0)
HLOCKx : 要求完全的匯流排使用權

Address decoding

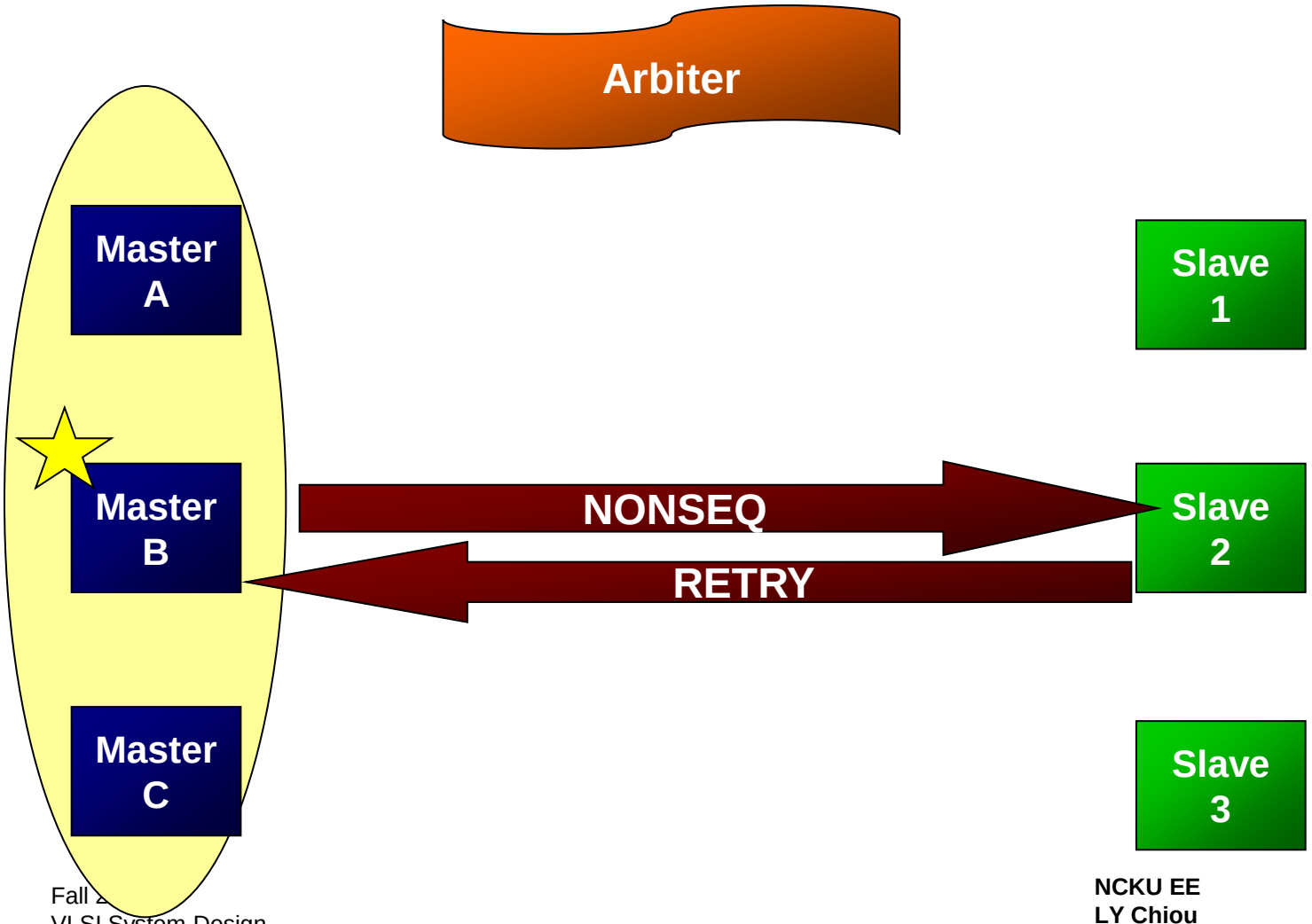


Slave --- transfer response

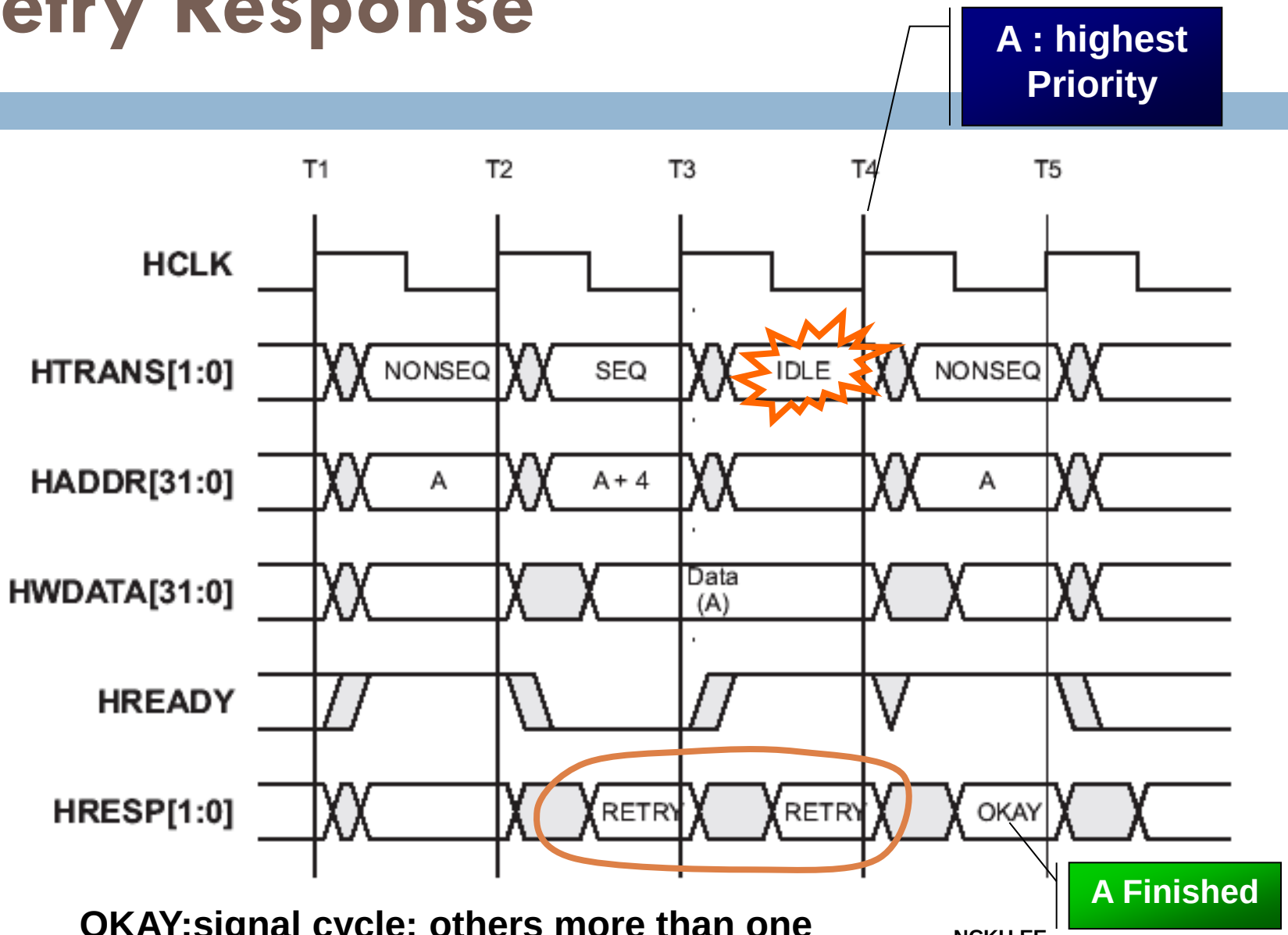


HREADY: extend transfer, end
HRESP ：Slave結束時的status

RETRY

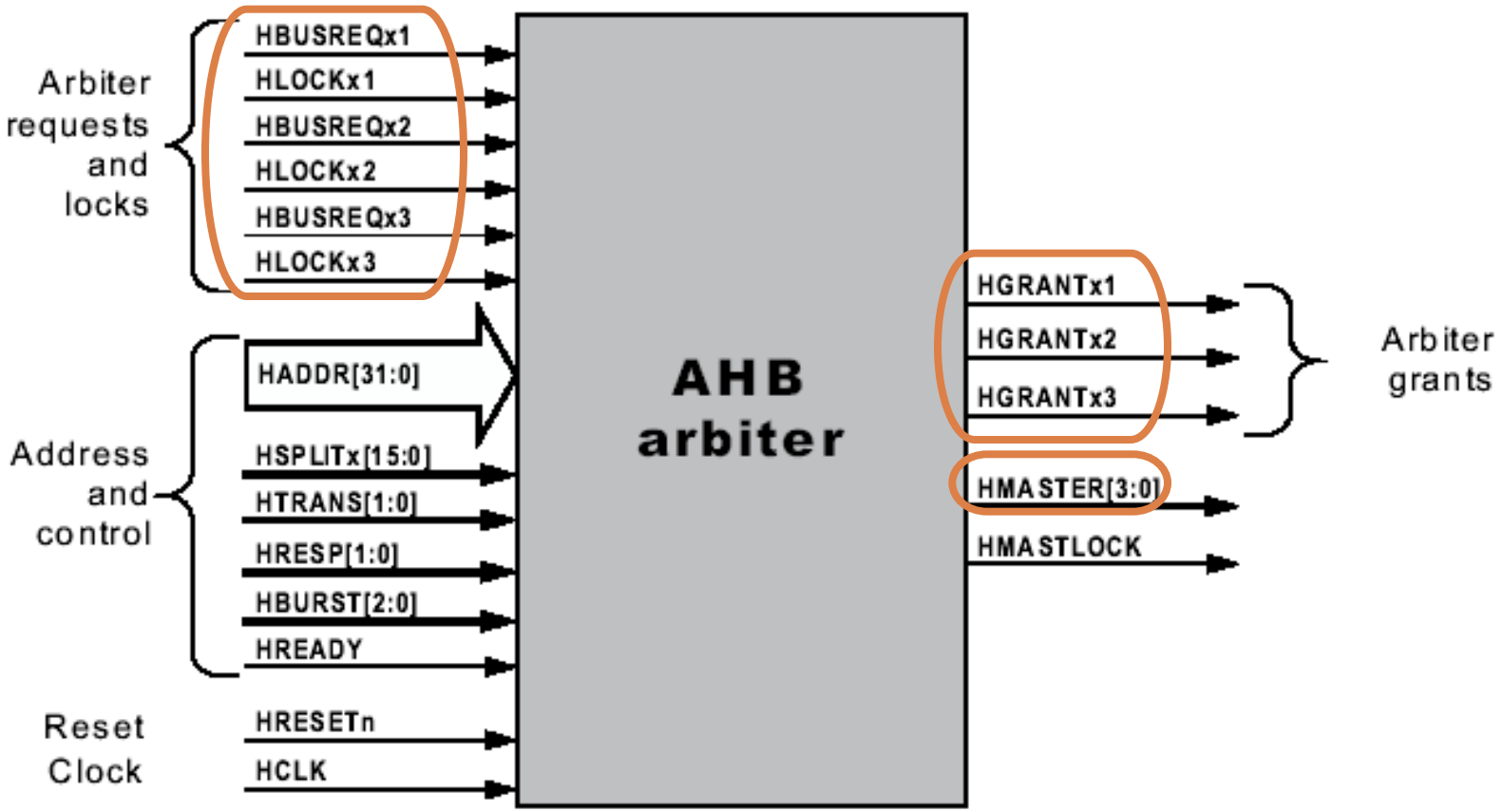

Arbiter
Master A
Master B
Master C
Slave 1
Slave 2
Slave 3
NONSEQ
RETRY

Retry Response

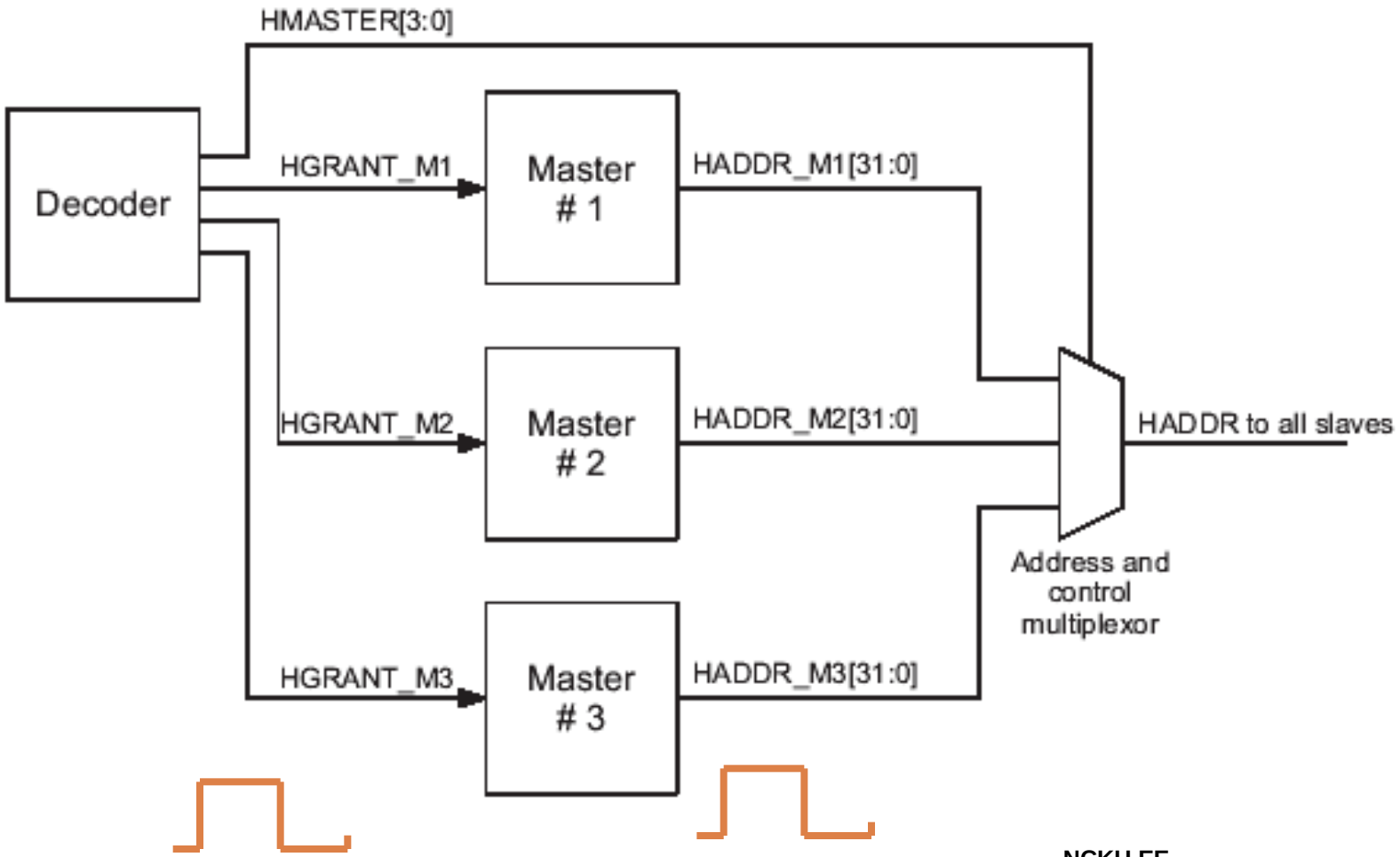


OKAY:signal cycle; others more than one

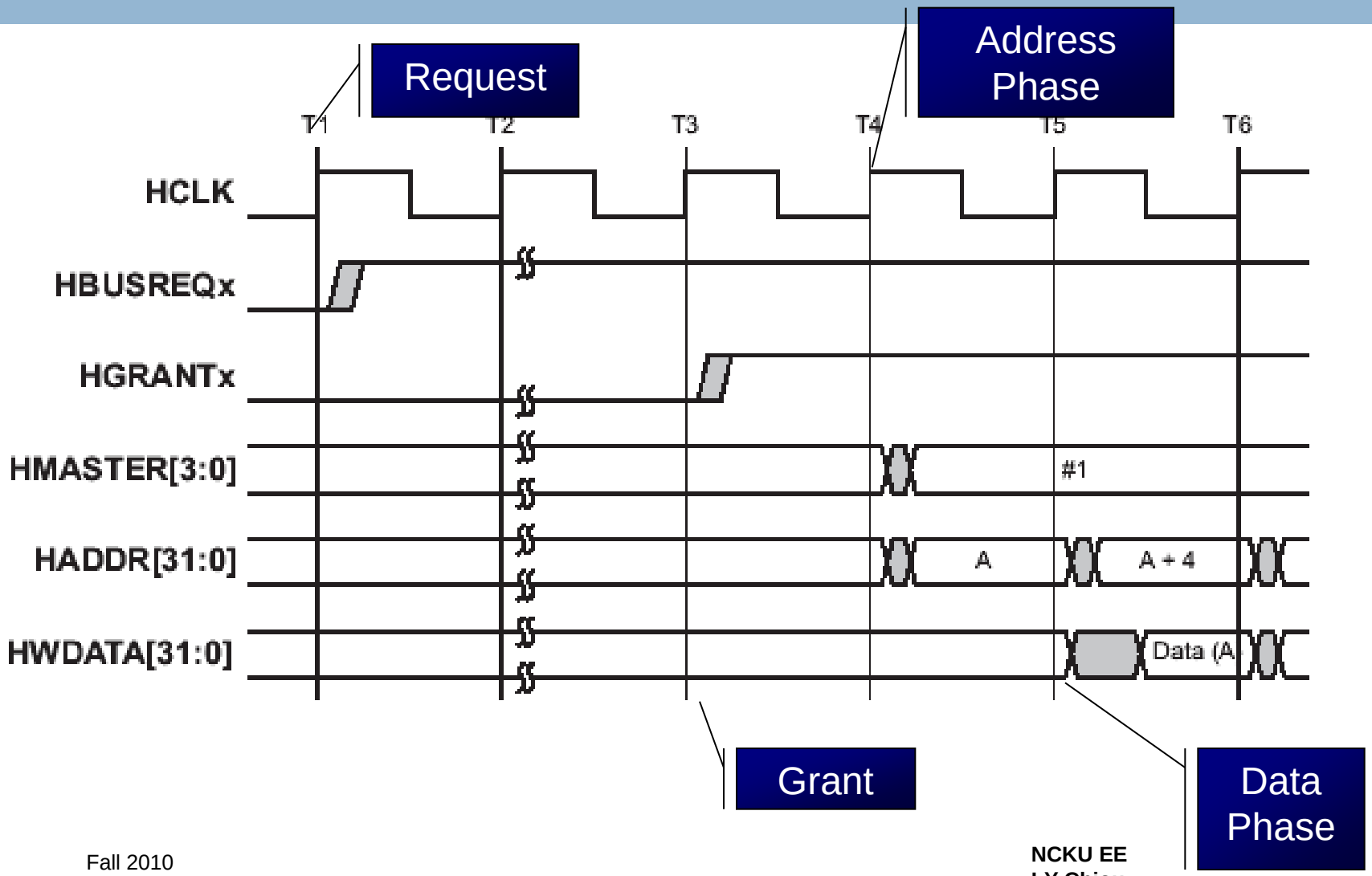
Arbiter


Arbiter requests and locks
HBUSREQx1
HLOCKx1
HBUSREQx2
HLOCKx2
HBUSREQx3
HLOCKx3
Address and control
HADDR[31:0]
HSPLITx[15:0]
HTRANS[1:0]
HRESP[1:0]
HBURST[2:0]
HREADY
Reset Clock
HRESETn
HCLK
AHB arbiter
HGRANTx1
HGRANTx2
HGRANTx3
Arbiter grants
HMASTER[3:0]
HMASTLOCK

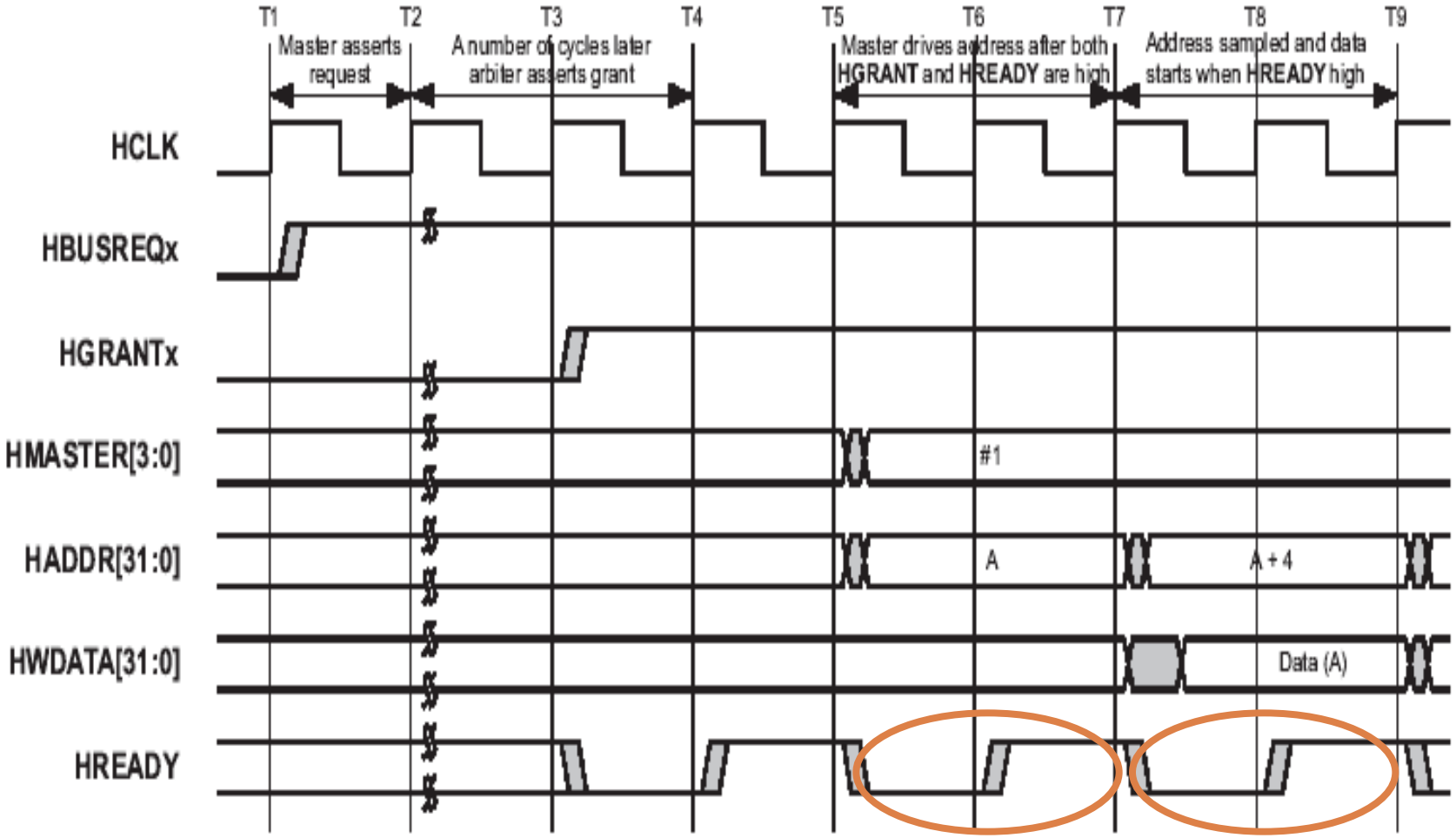
Bus master grant signals



Arbiter --- Grant access with no wait state



Arbiter --- Grant access with wait state

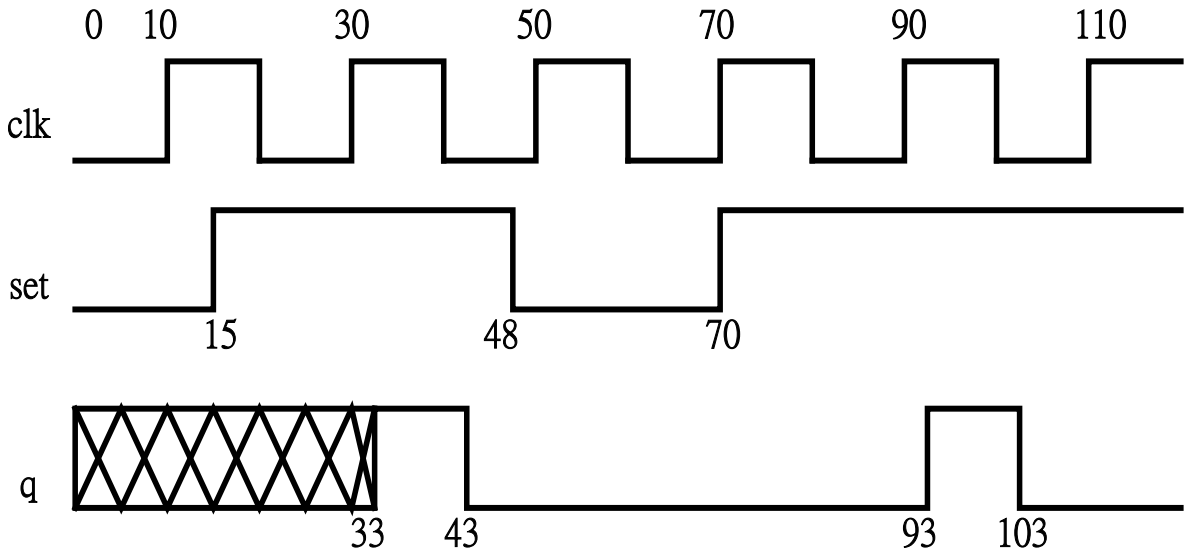


Timing Control in Procedural Blocks

- Three types of timing controls.
  - **Simple Delay**
    - **#10 rega = regb;**
    - **#(cycle/2) clk = ~clk; //cycle is declared as parameter**
  - **Edge-Trigger Timing Control**
    - **@(r or q) rega = regb; // controlled by "r" or "q"**
    - **@(posedge clk) rega = regb; // controlled by positive edge**
    - **@(negedge clk) rega = regb; // controlled by negative edge**
  - **Level-Triggered Timing Control**
    - **wait (!enable) rega = regb; // will wait until enable=0**

Timing Control in Procedural Blocks (continued)

```
always wait(set)
Begin @(posedge clk)
      #3 q = 1;
      #10 q = 0;
      wait(!set);
end
```



Read Task and Write Task

- Read
  - parameters: haddr, hsize, hburst
- Read(32'h48, `Hsize_Word, `Hburst_WRAP4)
  - address is 48, in word, burst by WRAP4
- Read(32'h64, `Hsize_Word, `Hburst_INCR8)
  - address is 48, in word, burst by INCR8

- Write
  - parameters: haddr, hsize, hburst, wdata
- Write(32'h48, `Hsize_Word, `Hburst_WRAP4, 32'ha0, 32'ha1,32'ha2, 32'ha3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
  - address is 48, in word, burst by WRAP4

```
`timescale 1ns/10ps
module ex3_1;

reg clk,nrst;
initial
begin
        clk=0;
        nrst=1;
  #45   nrst=0;
  #40   nrst=1;
end

always #5 clk=~clk;
//=====================================================
// AMBA parameter define
// Note! AMBA 定義 16 bits 叫做 Half-word
//                 32 bits 叫做 Word
//=====================================================
`define Hsize_Byte      3'b000
`define Hsize_Halfword  3'b001
`define Hsize_Word      3'b010

`define Hburst_SINGLE   3'b000
`define Hburst_INCR     3'b001
`define Hburst_WRAP4    3'b010
`define Hburst_INCR4    3'b011
`define Hburst_WRAP8    3'b100
`define Hburst_INCR8    3'b101

`define Hburst_INCR16   3'b111

`define SINGLE    3'b000
`define INCR      3'b001
`define WRAP4     3'b010
`define INCR4     3'b011
`define WRAP8     3'b100
`define INCR8     3'b101

`define INCR16    3'b111
//=====================================================
```

```
//=====================================================
// declaration
//=====================================================
reg write;
reg  [1:0]  trans;
reg  [31:0] addr;
reg  [2:0]  burst,rd_burst;
reg  [3:0]  addr_inc;
reg  [2:0]  size;
reg  [31:0] wdata;
reg  [4:0]  w_index;

//=====================================================
// slave signal declaration
//=====================================================
reg [1:0] resp;
reg write_en,read_en;
reg ready;
initial begin
 write_en=0;
 read_en=0;
end

initial begin
 trans=0;  //一開始是idle
 write=1;
 addr =0;
 burst=0;
 size =0;
 wdata=32'h0;
end

//---temp for slave's response
initial begin
   resp=0;
   ready=1;
end
```

```
initial
begin
//      for simulation
         Idle(10);

         //start simulation
         //Read(addr,hsize,length,burst_type)
         Read(32'h48,`Hsize_Word,`Hburst_WRAP4);
         Idle(5);
         Read(32'h48,`Hsize_Word,`Hburst_INCR4);
         Idle(1);
         Read(32'h50,`Hsize_Halfword,`Hburst_SINGLE);
         Idle(1);
         Read(32'h68,`Hsize_Halfword,`Hburst_WRAP8);
         Idle(1);
         Read(32'h6c,`Hsize_Halfword,`Hburst_WRAP4);
         Idle(1);
         Read(32'h60,`Hsize_Word,`Hburst_WRAP8);
         Idle(1);
         Read(32'h64,`Hsize_Word,`Hburst_INCR8);
         Idle(1);
         Read(32'h68,`Hsize_Word,`Hburst_INCR16);
         Idle(10);
         //Write
         Write(32'h48,`Hsize_Word,`Hburst_WRAP4,
                32'ha0,32'ha1,32'ha2,32'ha3,0,0,0,0,0,0,0,0,0,0,0,0 );
                         Idle(1);
         Write(32'h58,`Hsize_Word,`Hburst_INCR16,
                32'hb0,  32'hb1,32'hb2,32'hb3,
                32'hb4,  32'hb5,32'hb6,32'hb7,
                32'hb8,  32'hb9,32'hba,32'hbb,
                32'hbc,  32'hbd,32'hbe,32'hbf);

         Idle(10);
         Write(32'h00,`Hsize_Word,`Hburst_INCR16,
                32'hb0,  32'hb1,32'hb2,32'hb3,
                32'hb4,  32'hb5,32'hb6,32'hb7,
                32'hb8,  32'hb9,32'hba,32'hbb,
                32'hbc,  32'hbd,32'hbe,32'hbf);
         Idle(5);
         // read first 16 registers back
         Read(32'h00,`Hsize_Word,`Hburst_INCR16);
         Idle(5);
         Write(32'h1c,`Hsize_Word,`Hburst_WRAP4,
                32'he0,  32'he1,32'he2,32'he3,
```

```
            0,0,0,0,0,0,0,0,0,0,0,0);
        Idle(5);
        Read(32'h1c,`Hsize_Word,`Hburst_WRAP4);

        Write(32'h04,`Hsize_Word,`Hburst_WRAP8,
            32'hf0,32'hf1,32'hf2,32'hf3,
            32'hf4,32'hf5,32'hf6,32'hf7,
            0,0,0,0,0,0,0,0);
        Idle(1);
        Read(32'h04,`Hsize_Word,`Hburst_WRAP8);
end
```

```
task Read;
input [31:0] haddr;
input [2:0]  hsize;  //傳 Byte ?
input [2:0]  hburst; //burst type

reg [4:0] i,rd_length;

begin
   read_en=1;
   //Transfer size decide the address increase control
   // see AMBA spec chap3.7.2 page3-17
   if (hsize==0)      addr_inc=1;
   else if (hsize==1) addr_inc=2;
   else if (hsize==2) addr_inc=4;

   burst=hburst;

   if (burst==`SINGLE)
           rd_length=1;
   else if ((burst==`WRAP4)|(burst==`INCR4))
           rd_length=4;
   else if ((burst==`WRAP8)|(burst==`INCR8))
           rd_length=8;
   else if (burst==`INCR16)
           rd_length=16;

   size=hsize;
   write=0;
   addr=#1 haddr;
   trans=2'b10;  // NONSEQ state
```

```
@(posedge clk);

//if (hburst>1)
trans=2'b11;  // SEQ state

wait(ready==1);
rd_length=rd_length;
i=rd_length;

repeat(rd_length)
begin
  //---address increase control ----------
  if ((burst==`WRAP8)&(hsize==1)) begin
   if (i==1) begin           addr=#1 0; trans=2'b00; end//addr;
   else if (addr[3:0]==4'he)addr=#1 addr-4'he;
   else                     addr=#1 addr+addr_inc;

  end else if ((burst==`WRAP8)&(hsize==2)) begin
   if (i==1) begin           addr=#1 0; trans=2'b00; end//addr;
   else if (addr[4:0]==5'h1c)addr=#1 addr-8'h1c;
   else                     addr=#1 addr+addr_inc;

  end else if ((burst==`WRAP4)&(hsize==1)) begin
   if (i==1) begin           addr=#1 0; trans=2'b00; end//addr;
   else if (addr[3:0]==4'he)addr=#1 addr-4'he;
   else                     addr=#1 addr+addr_inc;

  end else if ((burst==`WRAP4)&(hsize==2)) begin
   if (i==1) begin           addr=#1 0; trans=2'b00; end//addr;
   else if (addr[3:0]==4'hc)addr=#1 addr-4'hc;
   else                     addr=#1 addr+addr_inc;
```

```
      end else if ((burst==`SINGLE)|(burst==`INCR4)|
                   (burst==`INCR8)|(burst==`INCR16)) begin
      if (i==1) begin addr=#1 0; trans=2'b00; end
       else             addr=#1 addr+addr_inc;
      end

       wait(ready==1);
       @(posedge clk);
       i=i-1;
    end //end repeat
    if (resp==0) //Slave response OKAY
    Release_bus;
end
endtask
```

```
//=====================================================
// AHB Master write task
//=====================================================
task Write;
input [31:0] haddr;
input [2:0]  hsize;  //傳?Byte
input [2:0]  hburst; //burst type

input [31:0] hwdata1,hwdata2,hwdata3,hwdata4;
input [31:0] hwdata5,hwdata6,hwdata7,hwdata8;
input [31:0] hwdata9,hwdata10,hwdata11,hwdata12;
input [31:0] hwdata13,hwdata14,hwdata15,hwdata16;

reg [4:0] i,wr_length;

begin
   write_en=1;
   //Transfer size decide the address increase control
   // see AMBA spec chap3.7.2 page3-17
   if (hsize==0)      addr_inc=1;
   else if (hsize==1) addr_inc=2;
   else if (hsize==2) addr_inc=4;

   burst=hburst;

   if (burst==`SINGLE)
           wr_length=1;
   else if ((burst==`WRAP4)|(burst==`INCR4))
           wr_length=4;
   else if ((burst==`WRAP8)|(burst==`INCR8))
           wr_length=8;
   else if (burst==`INCR16)
           wr_length=16;

   size=hsize;
   write=1;
   addr=#1 haddr;
   trans=2'b10;  // NONSEQ state
```

```
@(posedge clk);

//if (hburst>1)
trans=2'b11;  // SEQ state

wait(ready==1);
wr_length=wr_length;
i=wr_length;
w_index=1;
wdata=#1 hwdata1;

repeat(wr_length)

begin
  //---address increase control ----------
  if ((burst==`WRAP8)&(hsize==1)) begin
   if (i==1) begin          addr=#1 0;//addr; !!!!要check 實際的電路
                                          //看最後一個cycle是不是還keep
   addr
                                          //或是addr必須改變
                            trans=2'b00;
             end
   else if (addr[3:0]==4'he)addr=#1 addr-4'he;
   else                     addr=#1 addr+addr_inc;

  end else if ((burst==`WRAP8)&(hsize==2)) begin
   if (i==1) begin           addr=#1 0; trans=2'b00; end //addr;
   else if (addr[4:0]==5'h1c)addr=#1 addr-8'h1c;
   else                      addr=#1 addr+addr_inc;
```

```
end else if ((burst==`WRAP4)&(hsize==1)) begin
 if (i==1) begin          addr=#1 0; trans=2'b00; end//addr;
 else if (addr[3:0]==4'he)addr=#1 addr-4'he;
 else                     addr=#1 addr+addr_inc;

end else if ((burst==`WRAP4)&(hsize==2)) begin
 if (i==1) begin          addr=#1 0; trans=2'b00; end//addr;
 else if (addr[3:0]==4'hc)addr=#1 addr-4'hc;
 else                     addr=#1 addr+addr_inc;

end else if ((burst==`SINGLE)|(burst==`INCR4)|
              (burst==`INCR8) |(burst==`INCR16)) begin
 if (i==1) begin addr=#1 0; trans=2'b00; end
 else            addr=#1 addr+addr_inc;
end
 wait(ready==1);
 @(posedge clk);
 i=i-1;
 w_index=w_index+1;
```

```
        if      (w_index==2)  wdata=#1 hwdata2;
        else if (w_index==3)  wdata=#1 hwdata3;
        else if (w_index==4)  wdata=#1 hwdata4;
        else if (w_index==5)  wdata=#1 hwdata5;
        else if (w_index==6)  wdata=#1 hwdata6;
        else if (w_index==7)  wdata=#1 hwdata7;
        else if (w_index==8)  wdata=#1 hwdata8;
        else if (w_index==9)  wdata=#1 hwdata9;
        else if (w_index==10) wdata=#1 hwdata10;
        else if (w_index==11) wdata=#1 hwdata11;
        else if (w_index==12) wdata=#1 hwdata12;
        else if (w_index==13) wdata=#1 hwdata13;
        else if (w_index==14) wdata=#1 hwdata14;
        else if (w_index==15) wdata=#1 hwdata15;
        else if (w_index==16) wdata=#1 hwdata16;

    end //end repeat
    if (resp==0) //Slave response OKAY
    Release_bus;
end
endtask
```