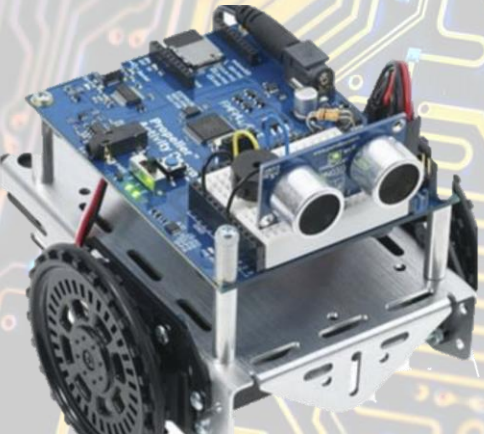
EC535 Introduction to Embedded Systems

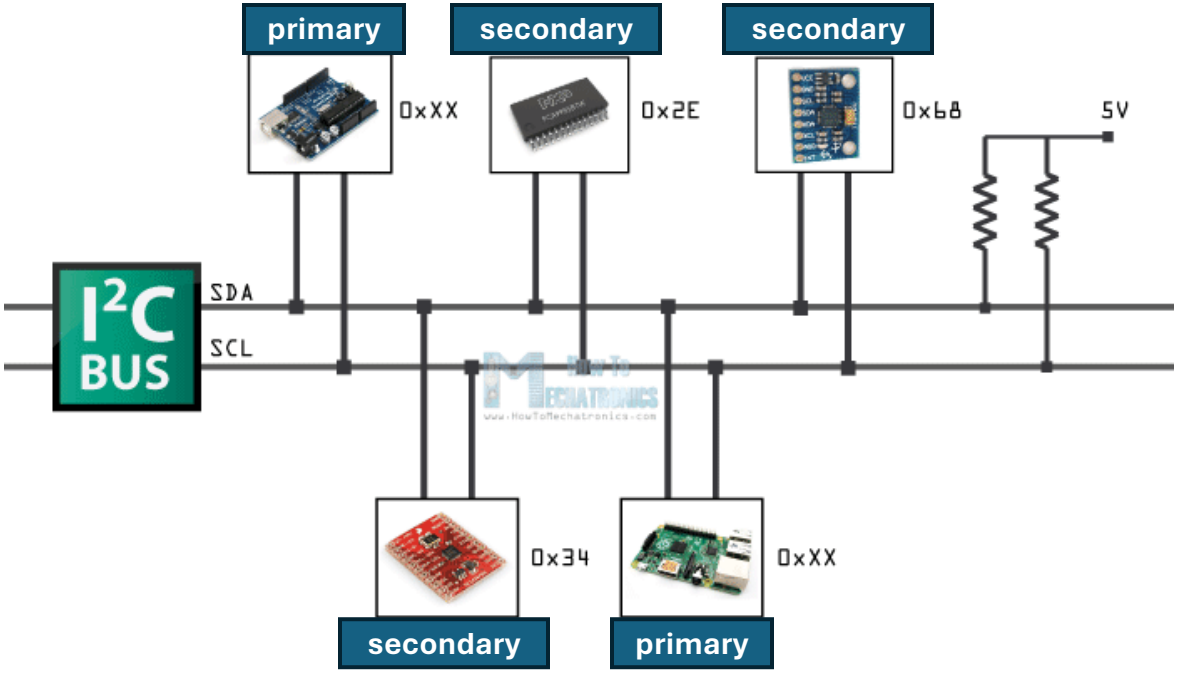


Exam Topics (4/15)

- Embedded System Design, Simulation
- Levels of Abstraction
- RTOS
- Scheduling
- Linux Device Drivers, Kernel Modules, Priorities
- Metrics, Power
- Blocking I/O, Asynchronous Notification
- Dataflow Modeling
- Bus Architectures
- Communication Protocols
- Memory

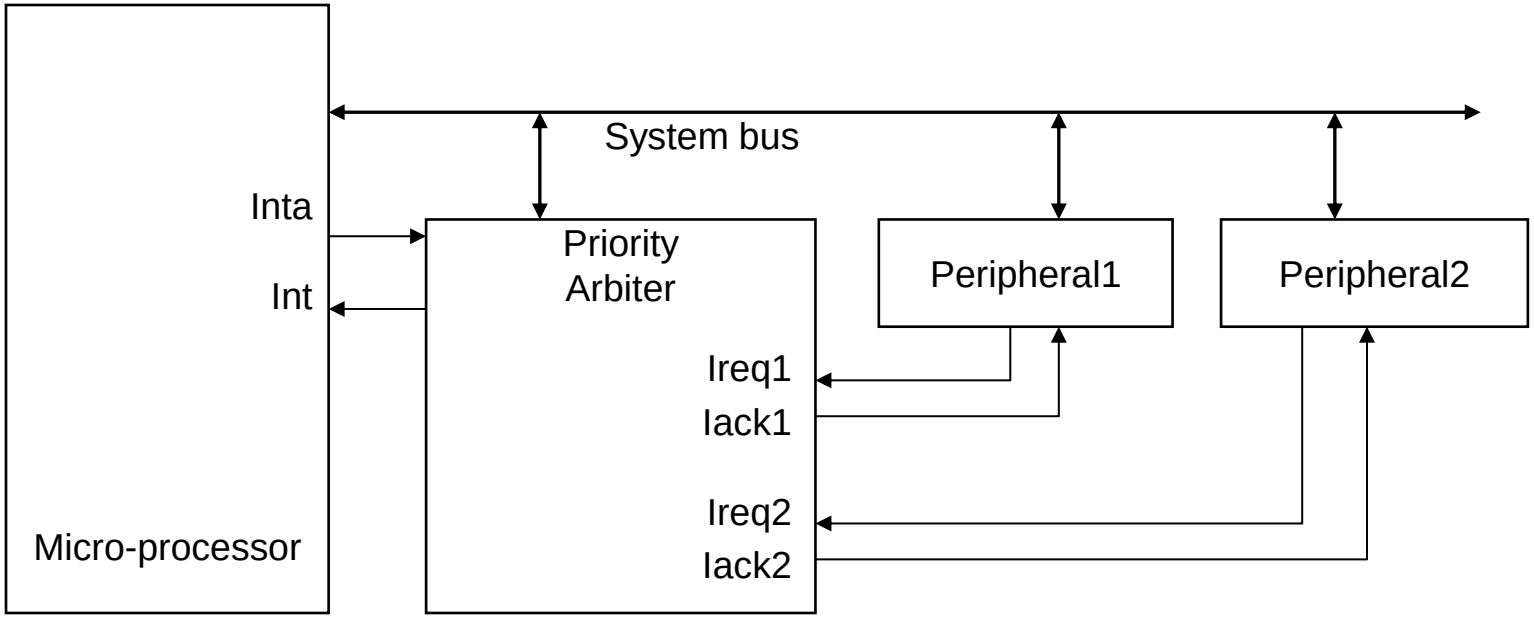
Serial protocols: I²C

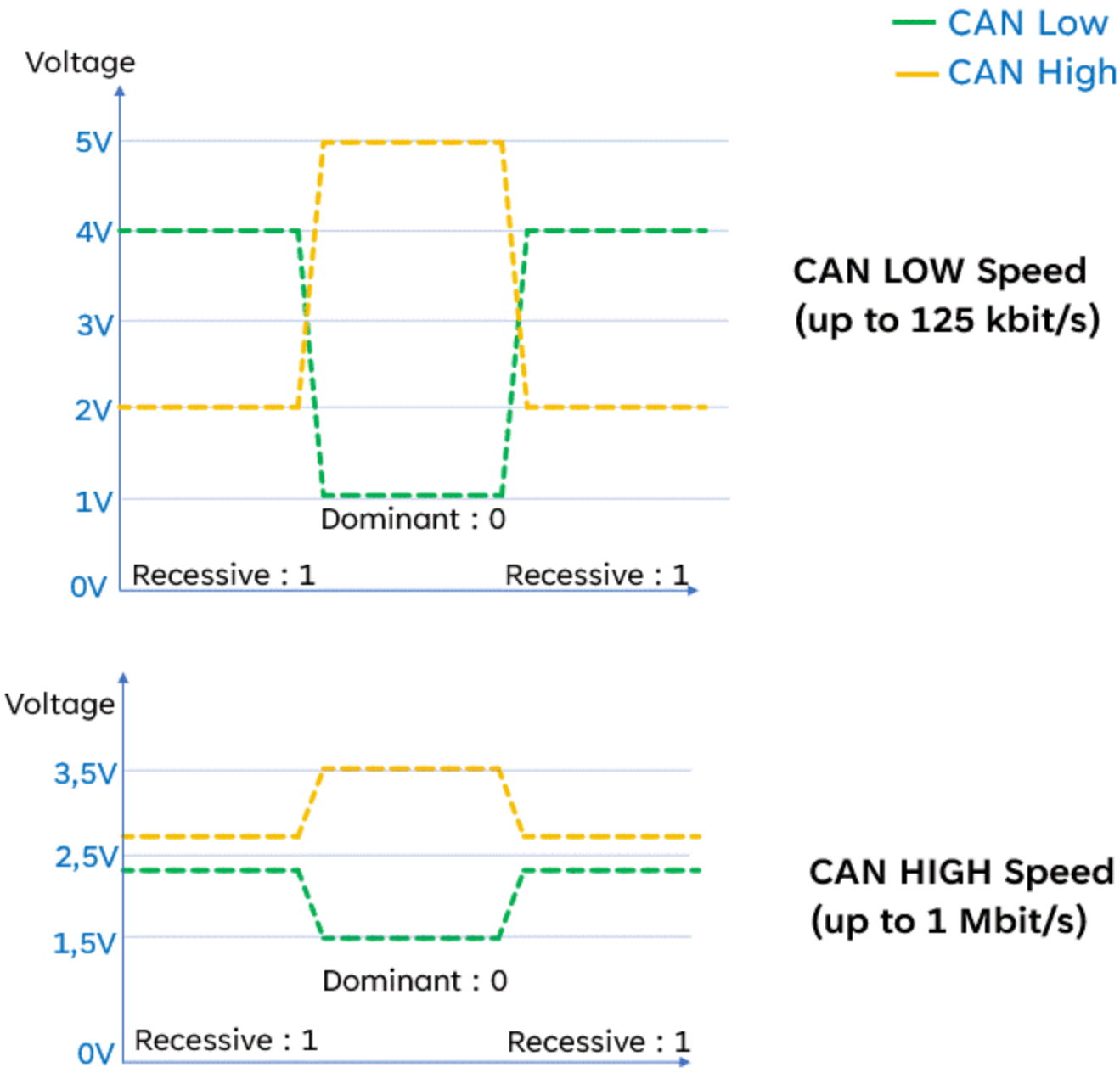
- I²C (Inter-IC)
  - Two-wire serial bus protocol originally developed by Philips
  - Data transfer rates up to 100 kbits/s and 7-bit addressing
  - Common devices capable of interfacing to I²C bus:
    - EPROMS, Flash, LCD controller, some RAM memory, real-time clocks, watchdog timers, and microcontrollers



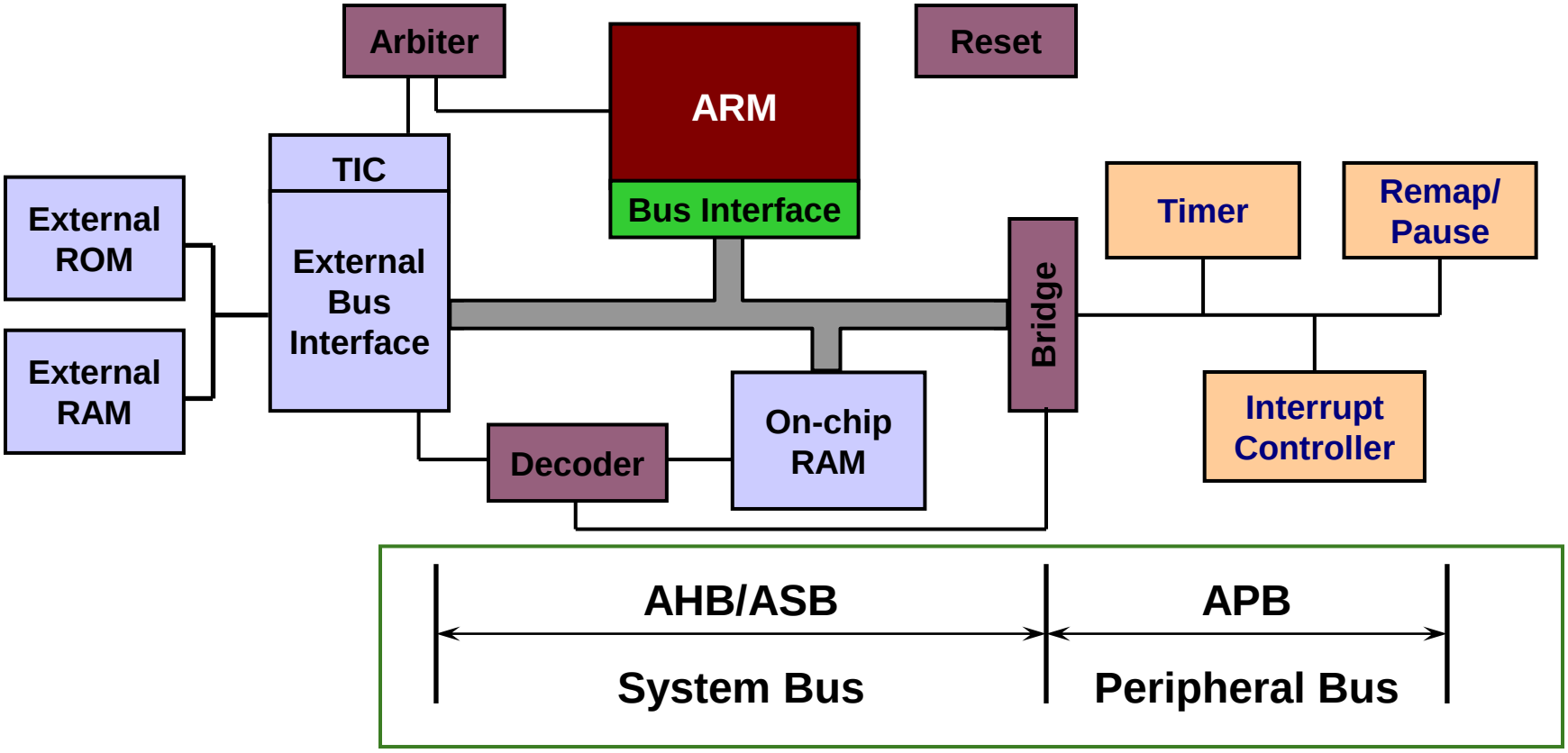
Arbitration: Priority arbiter

- Multiple peripherals request service from single resource (e.g., microprocessor, memory access controller) simultaneously.
- **Priority arbiter**
  - Single-purpose processor
  - Peripherals make requests to arbiter, arbiter makes requests to resource
  - Arbiter connected to system bus for ***configuration only***




CAN Low
CAN High
Voltage
5V
4V
3V
2V
1V
0V
Dominant : 0
Recessive : 1
Recessive : 1
CAN LOW Speed
(up to 125 kbit/s)
Voltage
3,5V
2,5V
1,5V
Dominant : 0
0V
Recessive : 1
Recessive : 1
CAN HIGH Speed
(up to 1 Mbit/s)

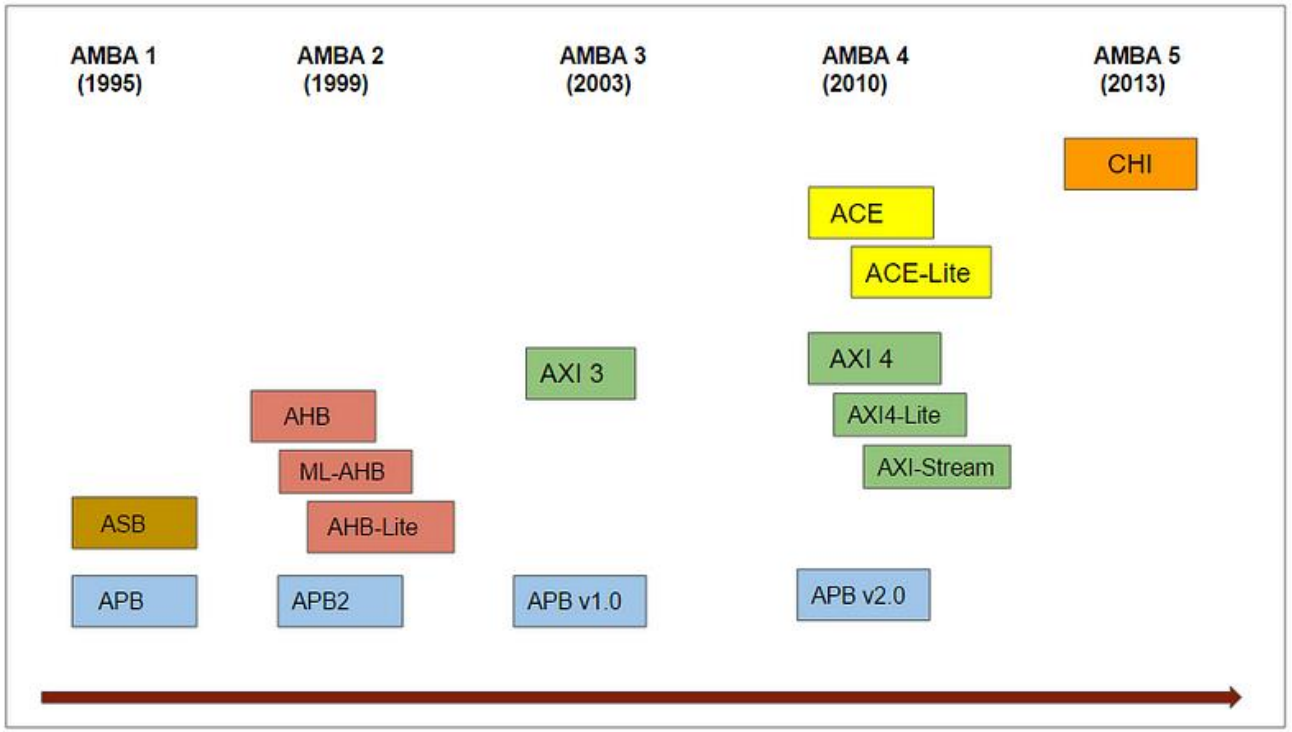
Typical ARM-based System



AMBA Protocol

A Parallel Protocol: AMBA Bus

- Advanced Microcontroller Bus Architecture (AMBA)
  - De facto standard in embedded processors
- AMBA is ARM's on-chip bus specification.



AMBA Sub-protocols

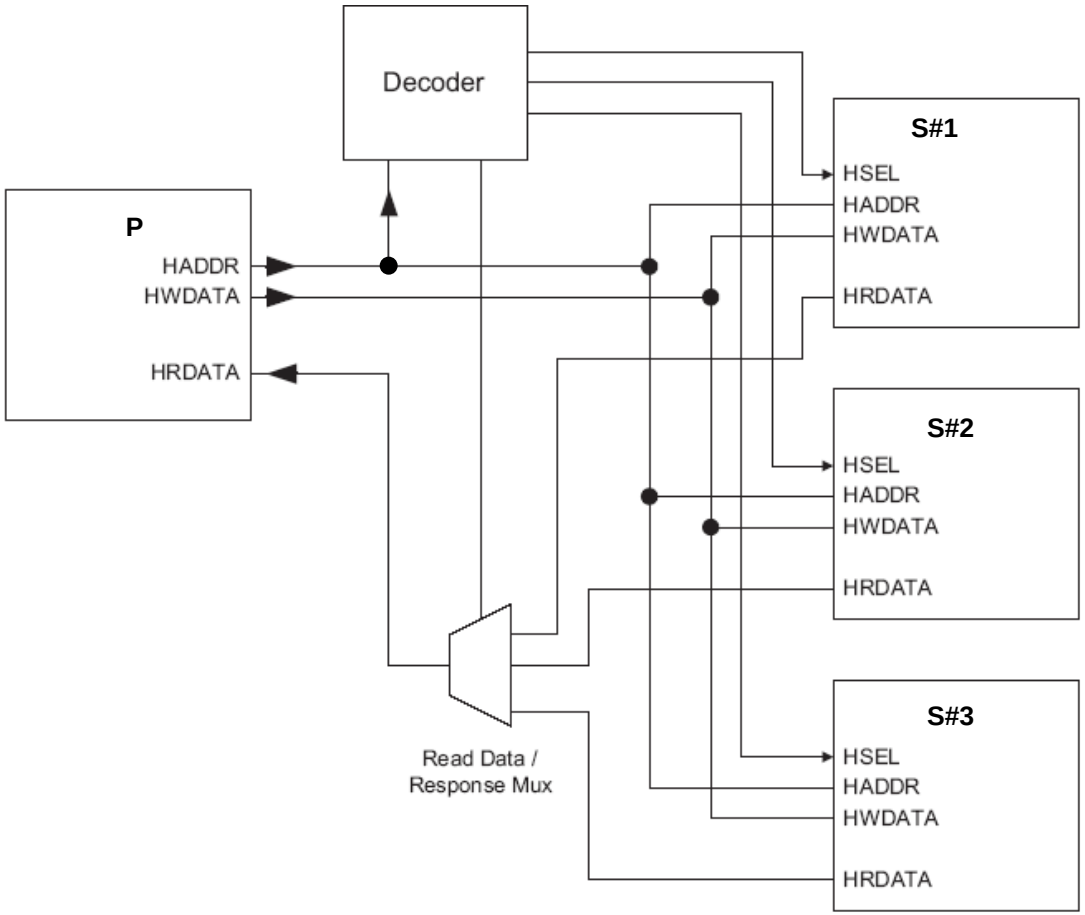
- **Advanced High-performance Bus (AHB)**
  - For high-performance, high clock frequency system modules.
  - All signal names start with H.
- **Advanced System Bus (ASB)**
  - Simplified version of AHB.
  - Most signal names start with B.
- **Advanced Peripheral Bus (APB)**
  - The AMBA APB is for low-power peripherals.
  - All signal names start with P.

High Performance

Minimal

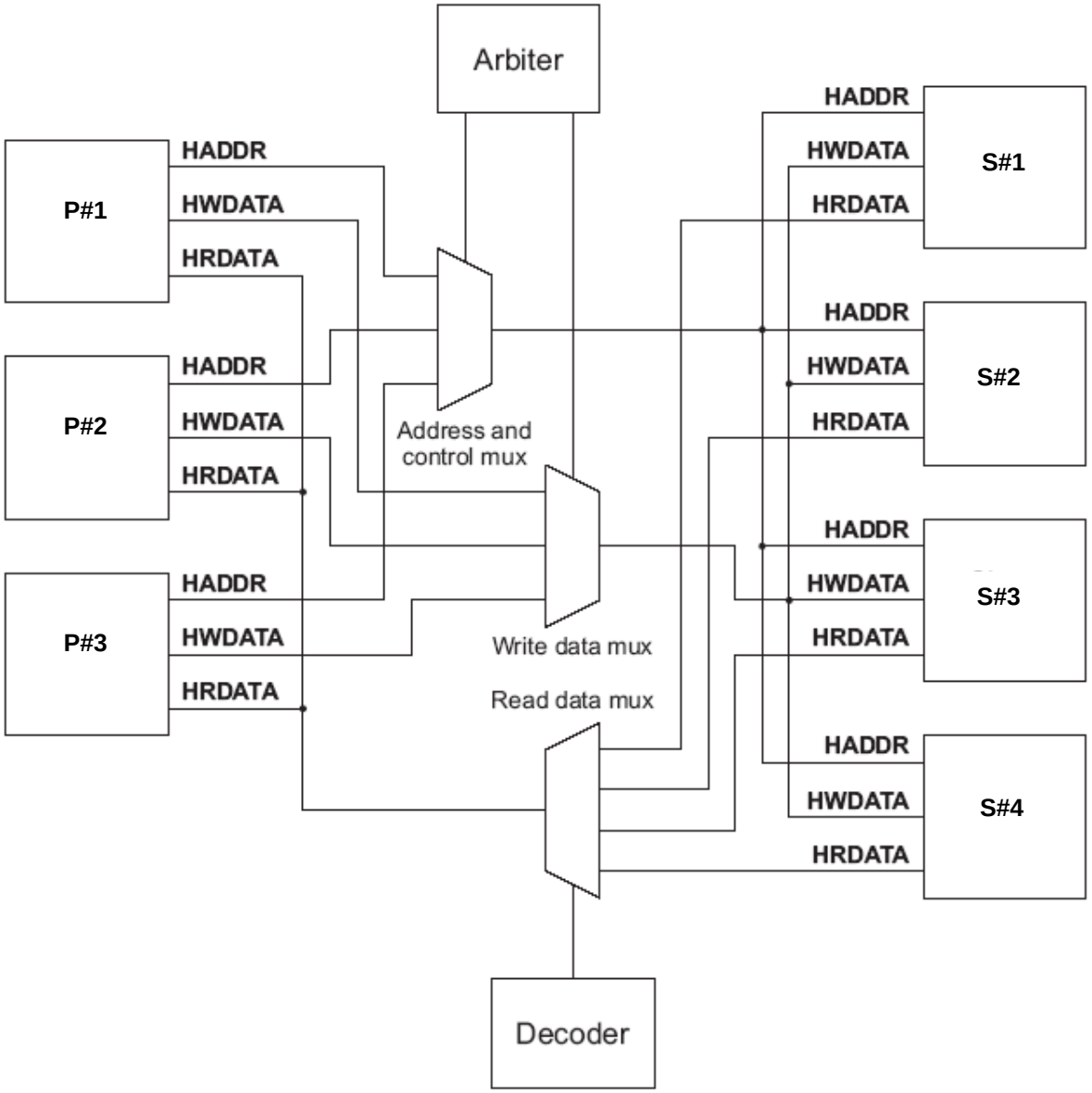
First, AHB-Lite

Single Primary: no arbiter needed



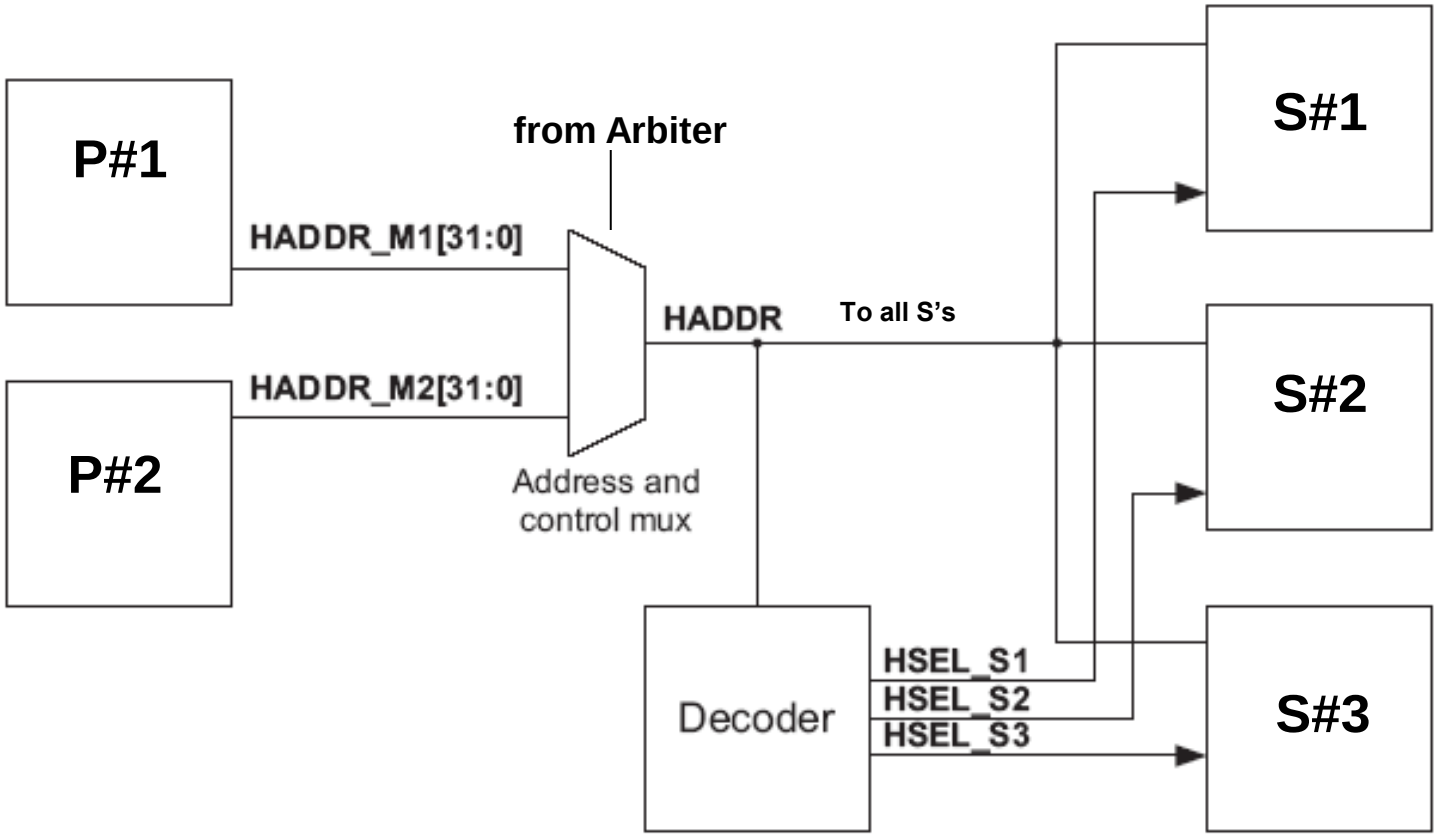
Full AHB

- Multiple primaries
- Arbiter controls address/data multiplexers



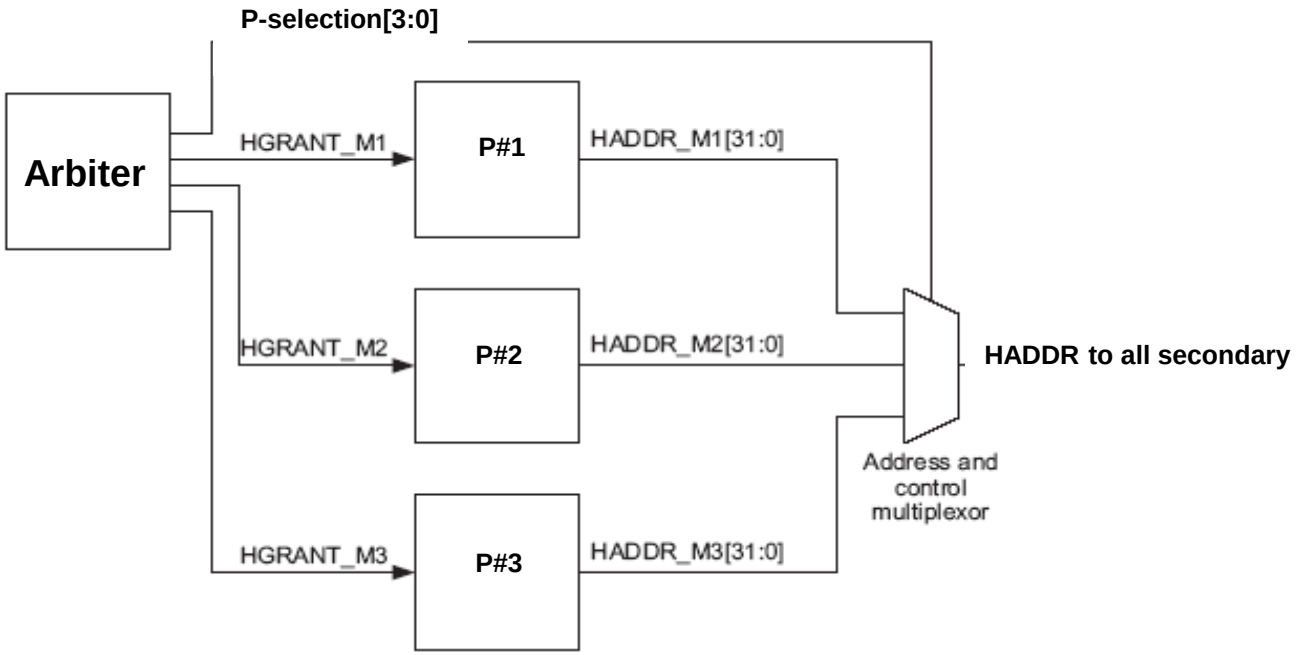
Address Decoding

- Decoder
  - Input: HADDR[31:0]
  - Output: HSELx



Arbitration

- Primary units send request to arbiter
- Arbiter grants access by asserting HGRANTx signals

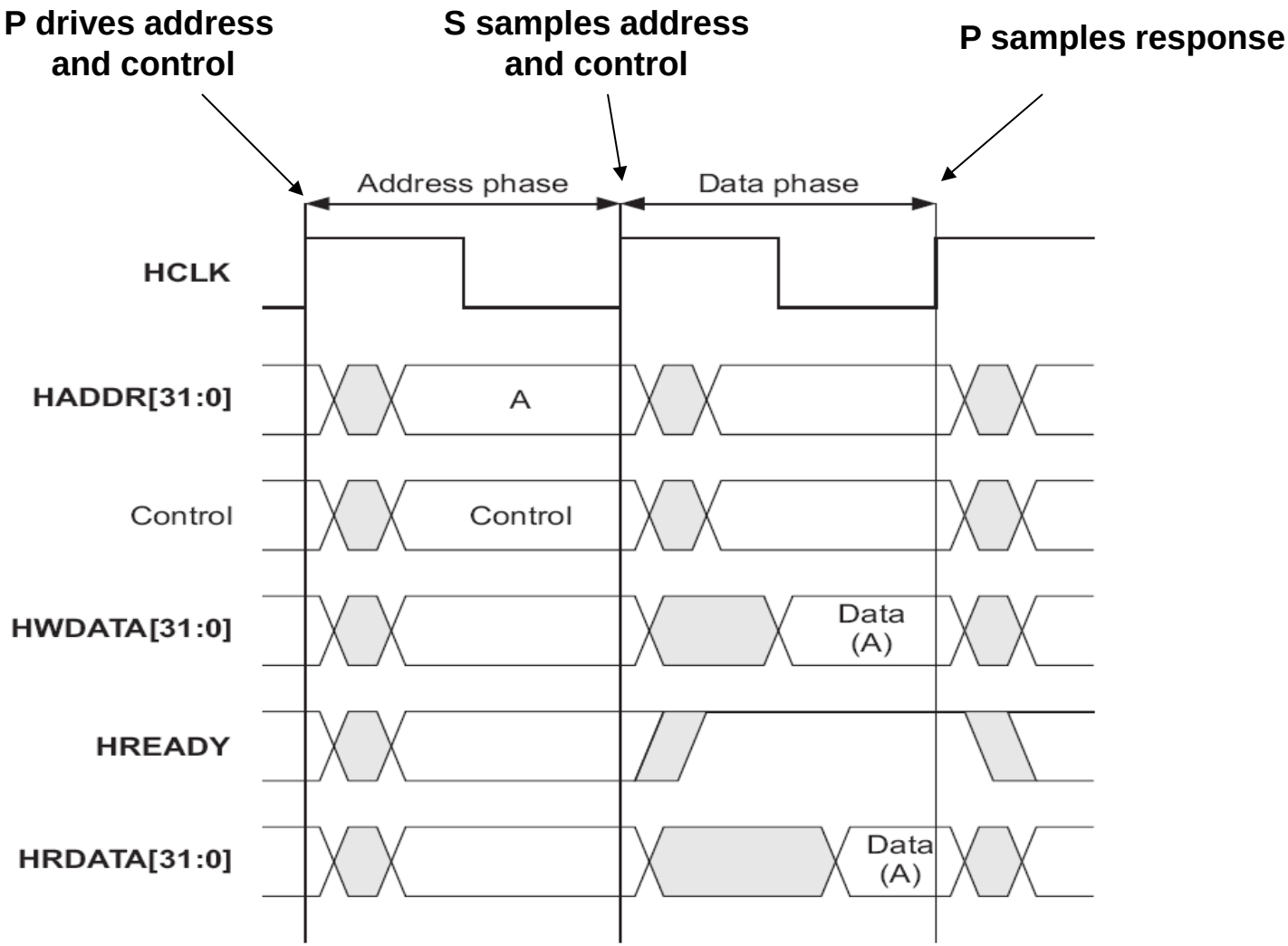


HBUSREQx from primary units not shown

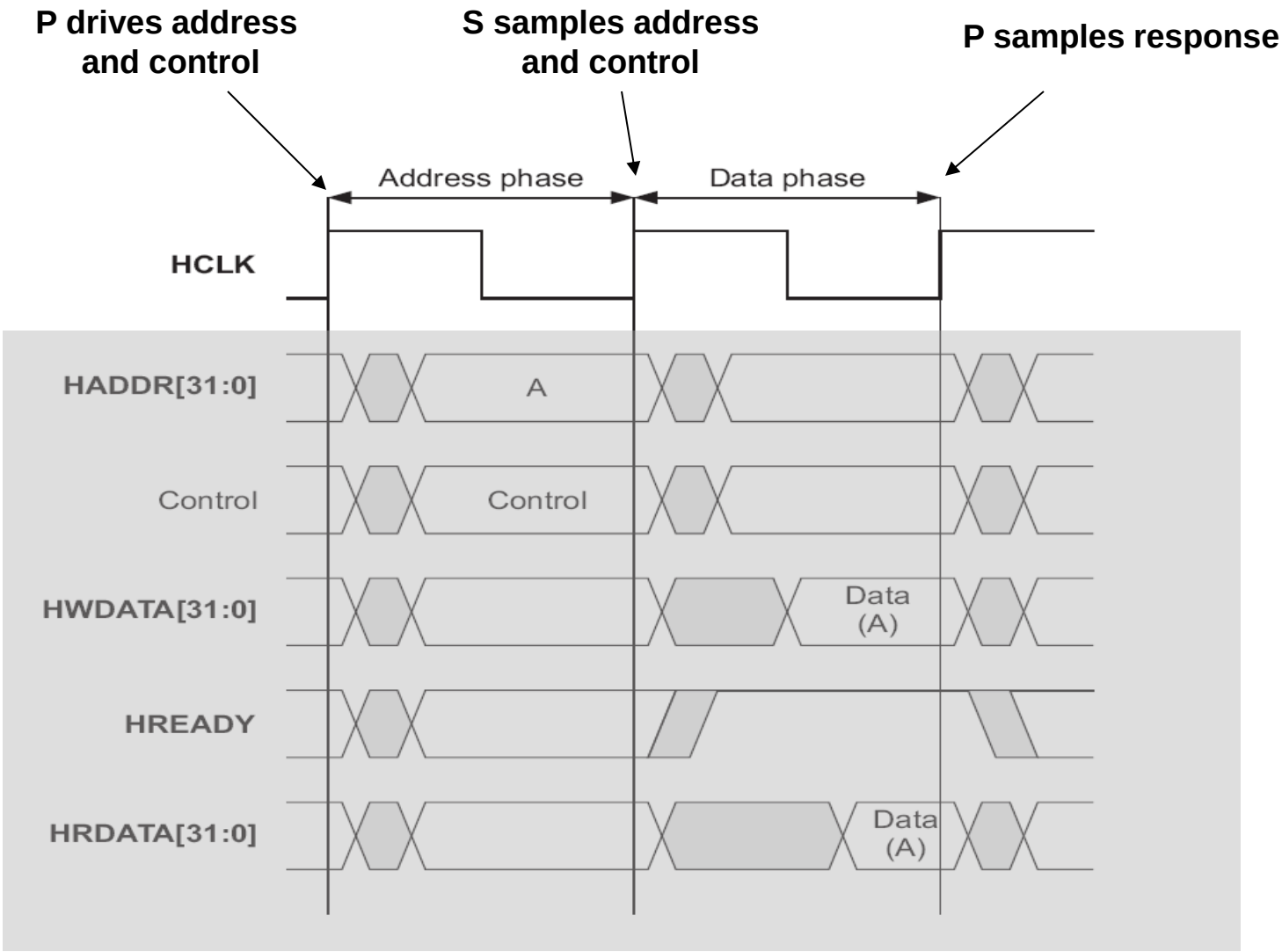
AHB Main Signals

name	source	
HCLK	Clock source	Global clock for all bus transactions
HADDR[31:0]	Primary	32bit address line
HWRITE	P	HI — write, LOW — read
HSIZE[2:0]	P	Transfer size (8bits – 1024bits)
HWDATA[31:0]	P	Data to write
HSELx	Decoder	Selects intended secondary
HREADY	Secondary	HI — a transfer finishes, LOW — wait
HRDATA[31:0]	S	Data to read by primary

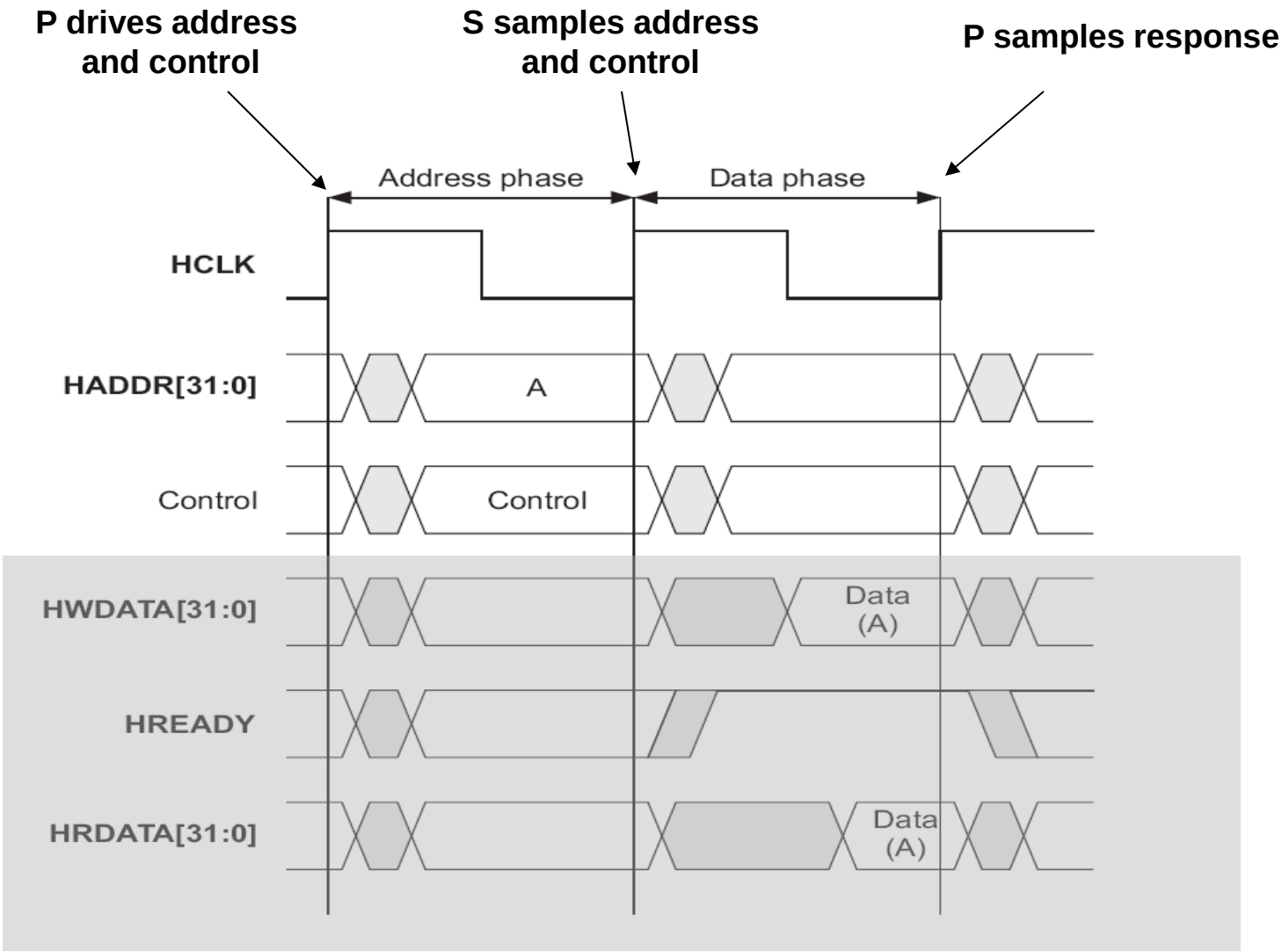
A Simple Transfer



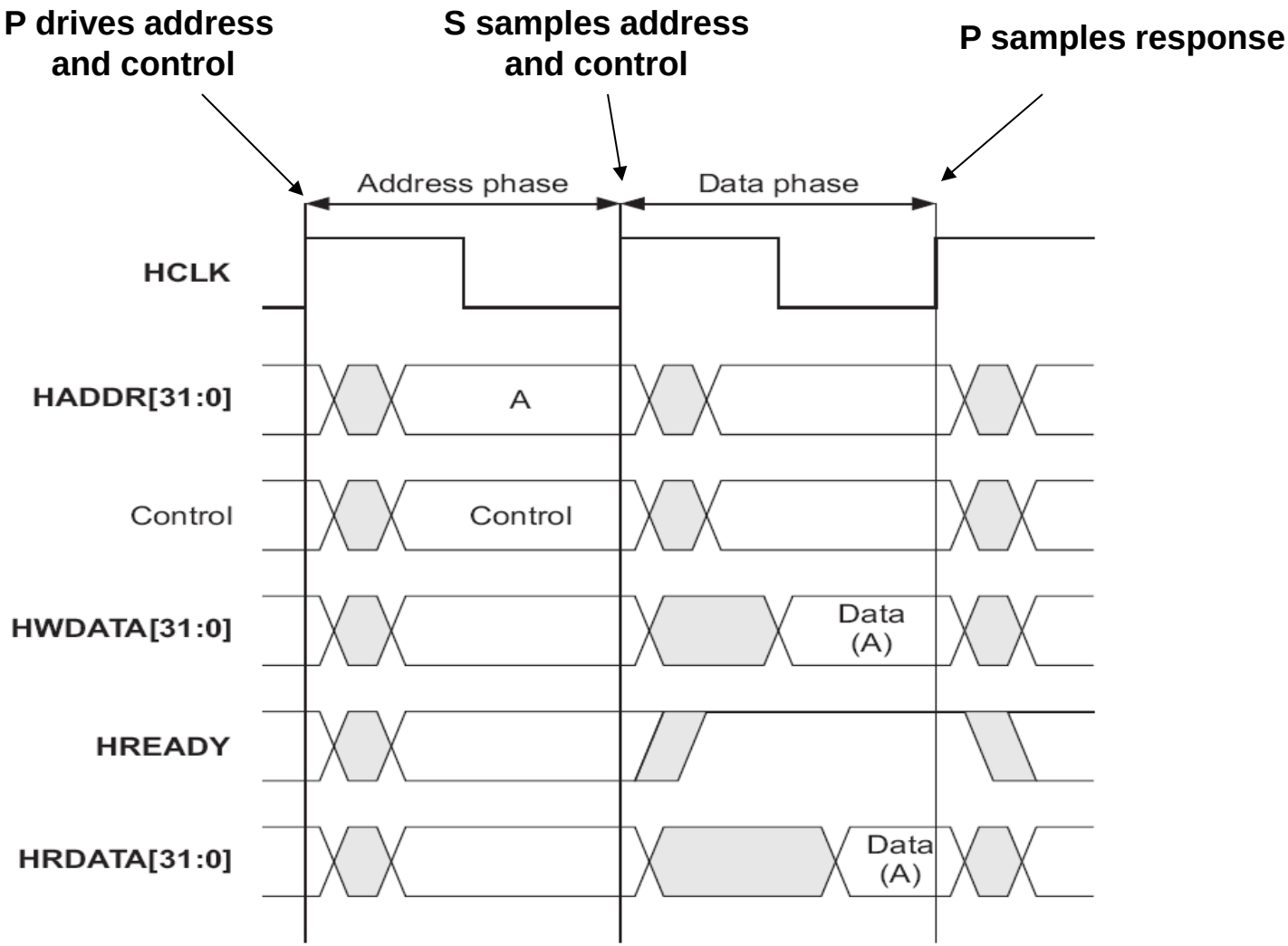
A Simple Transfer



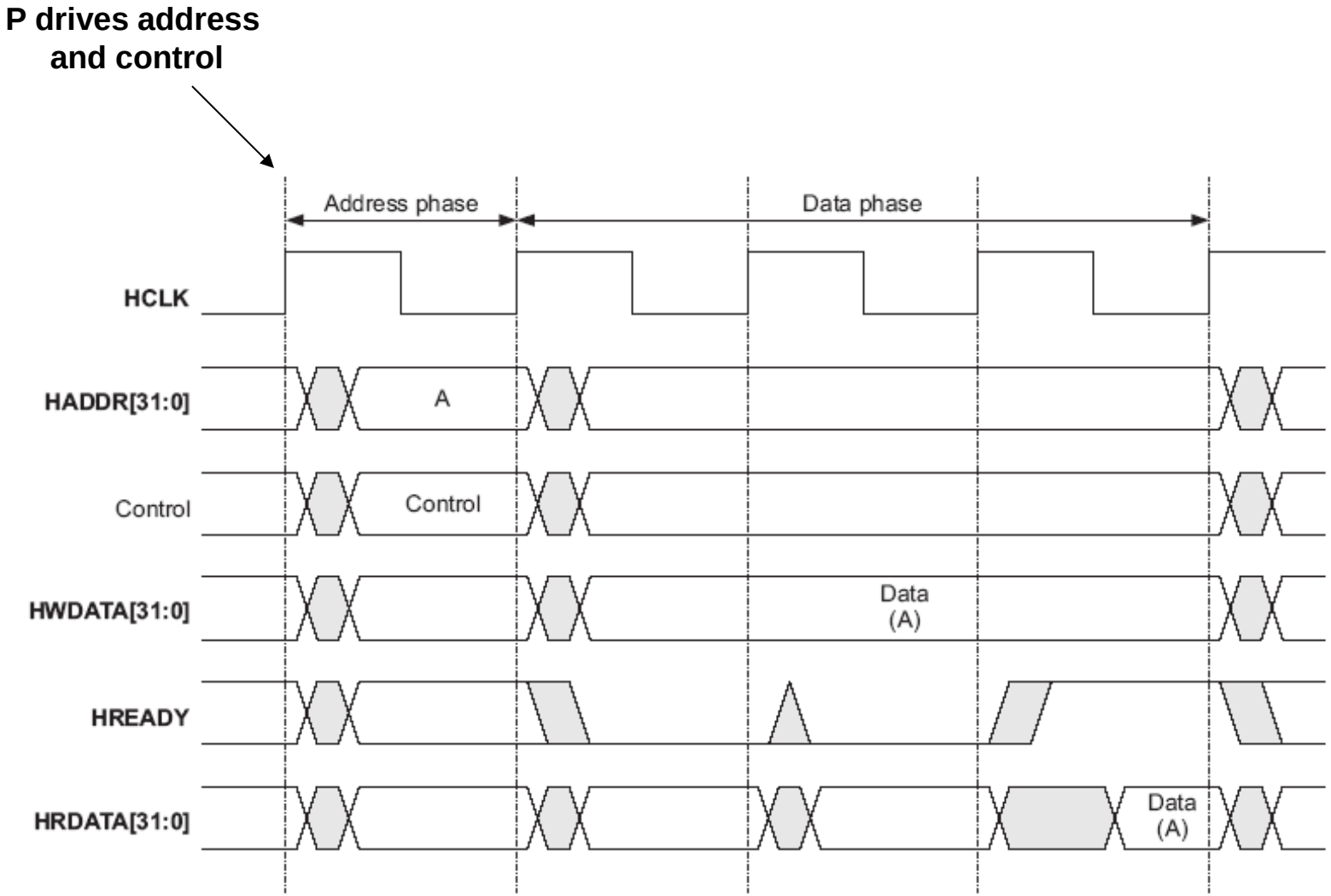
A Simple Transfer



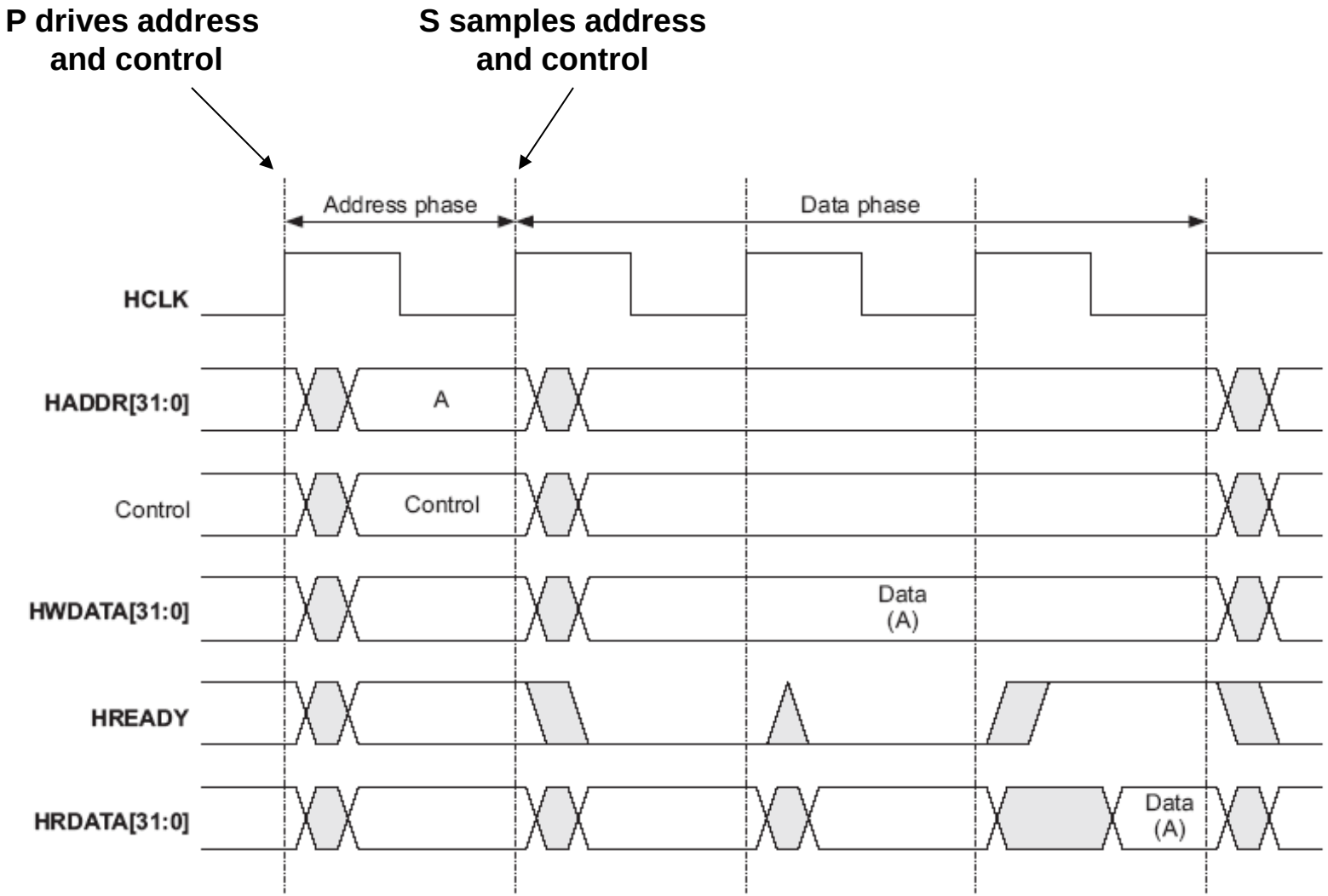
A Simple Transfer



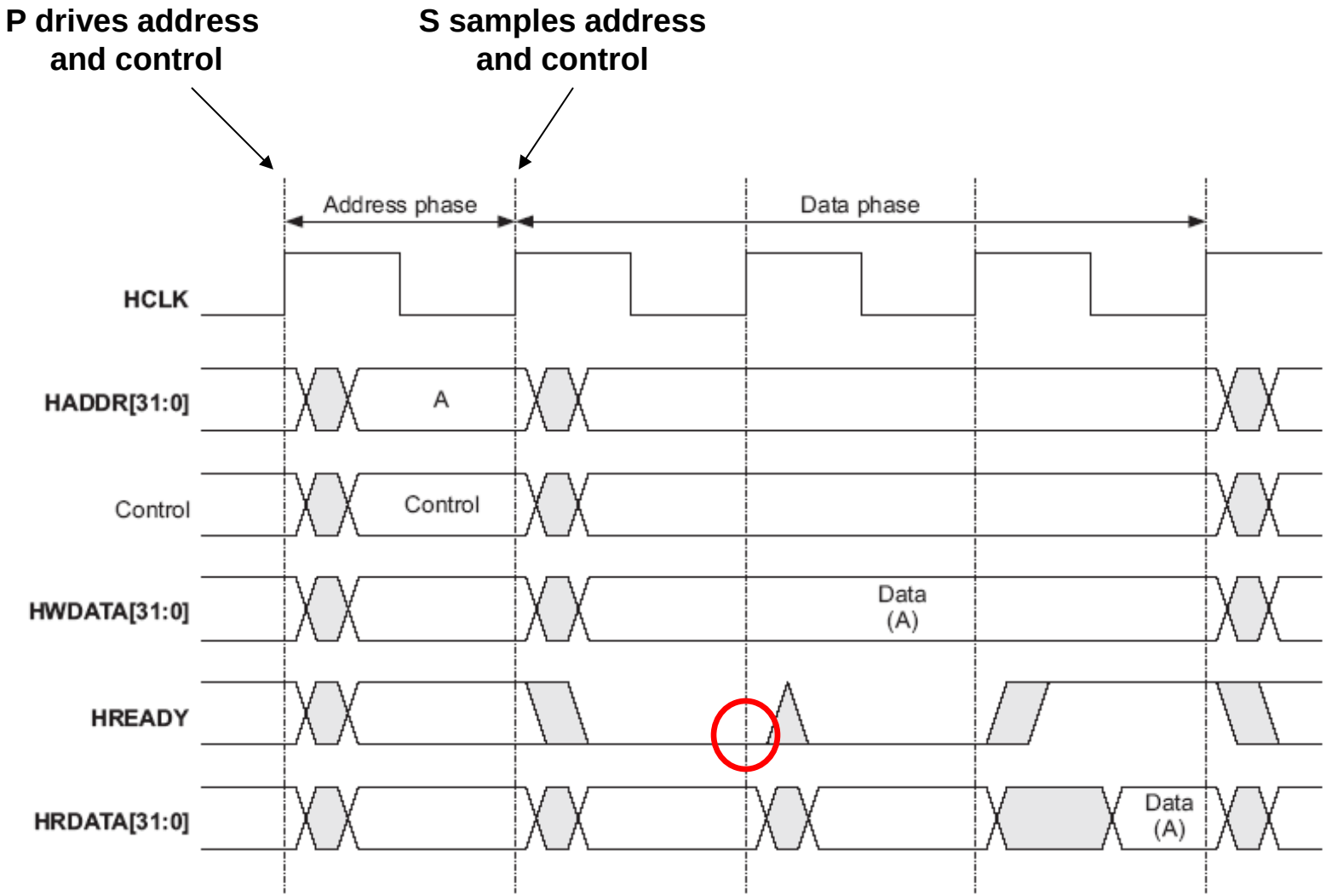
Transfer with Wait States



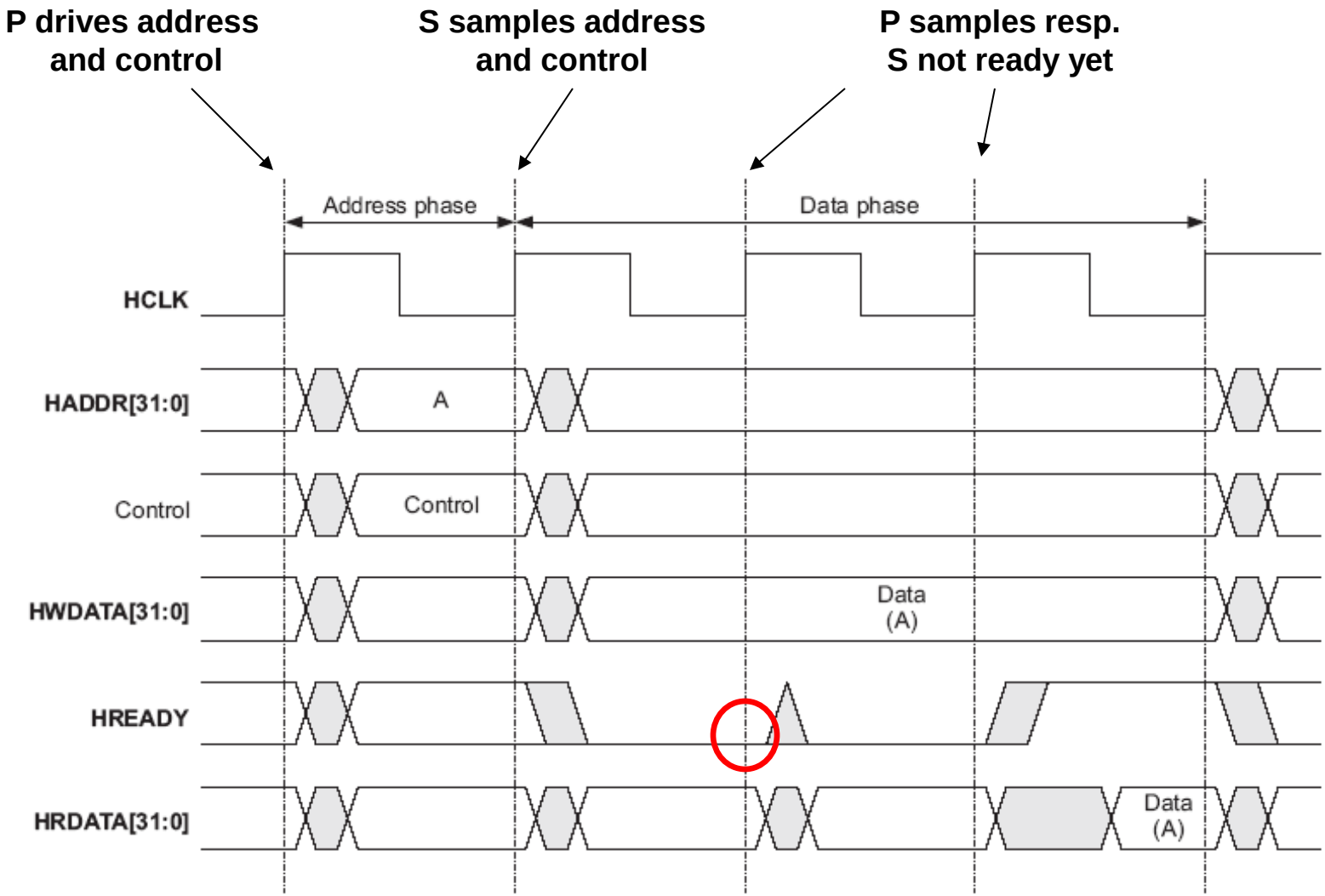
Transfer with Wait States



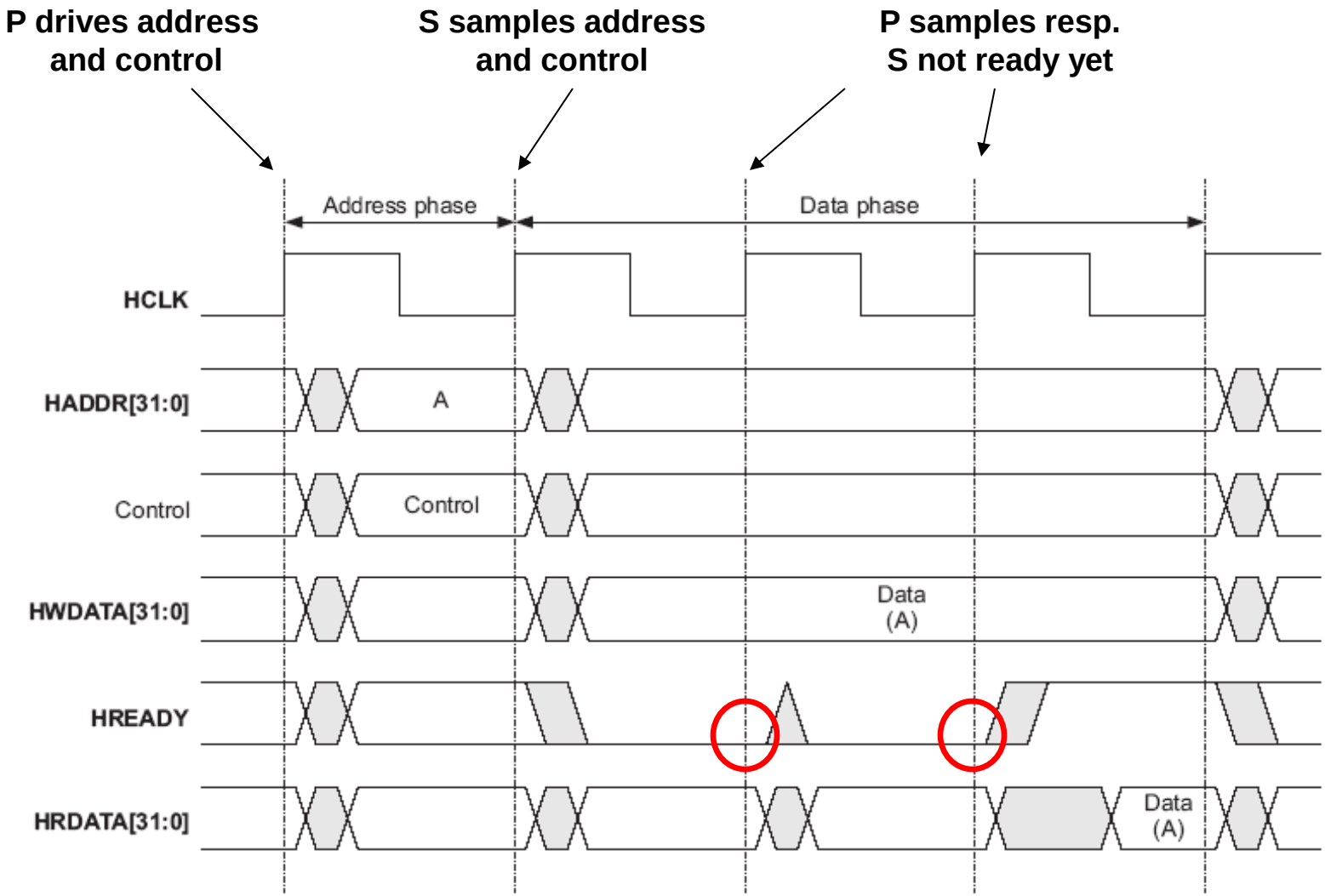
Transfer with Wait States



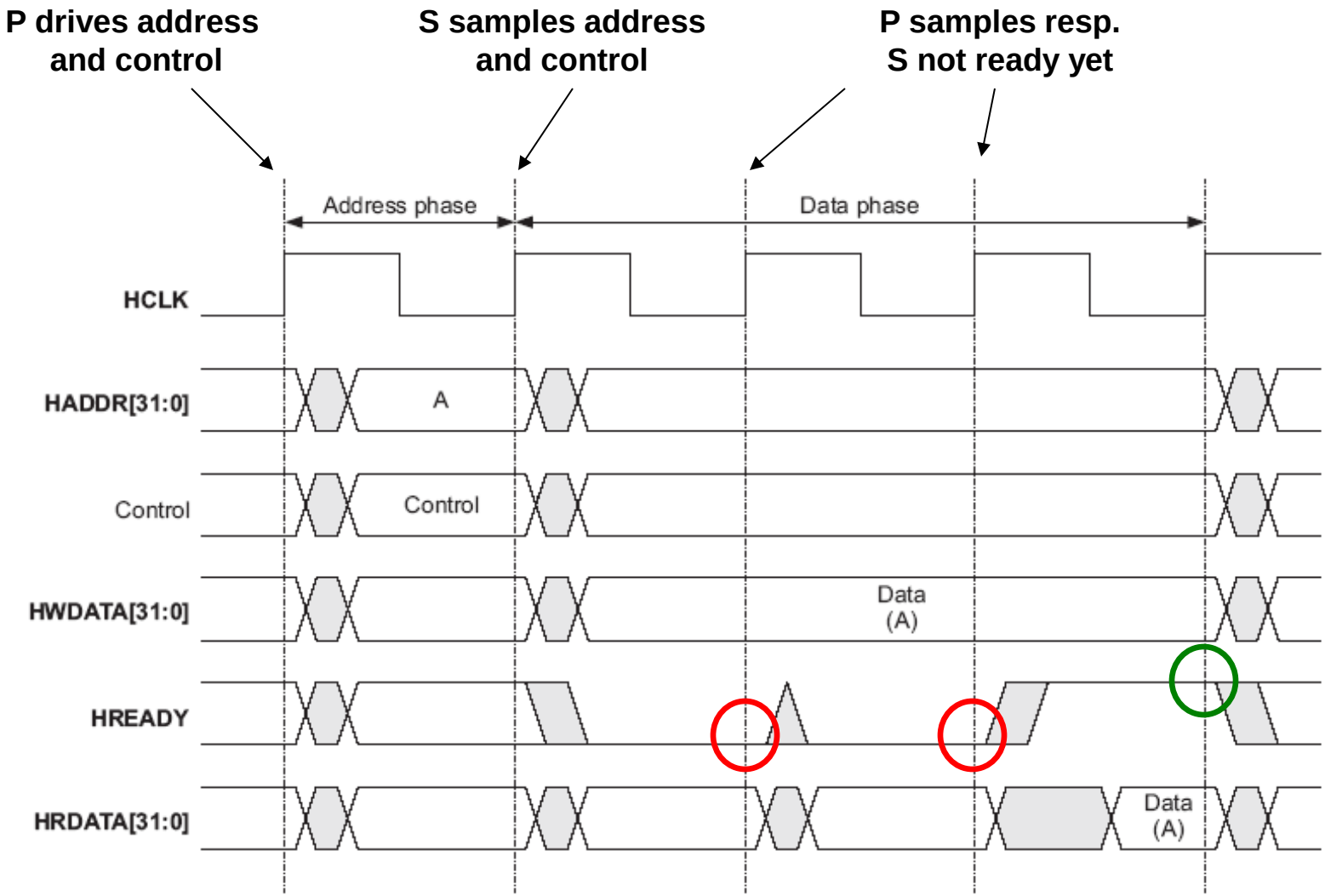
Transfer with Wait States



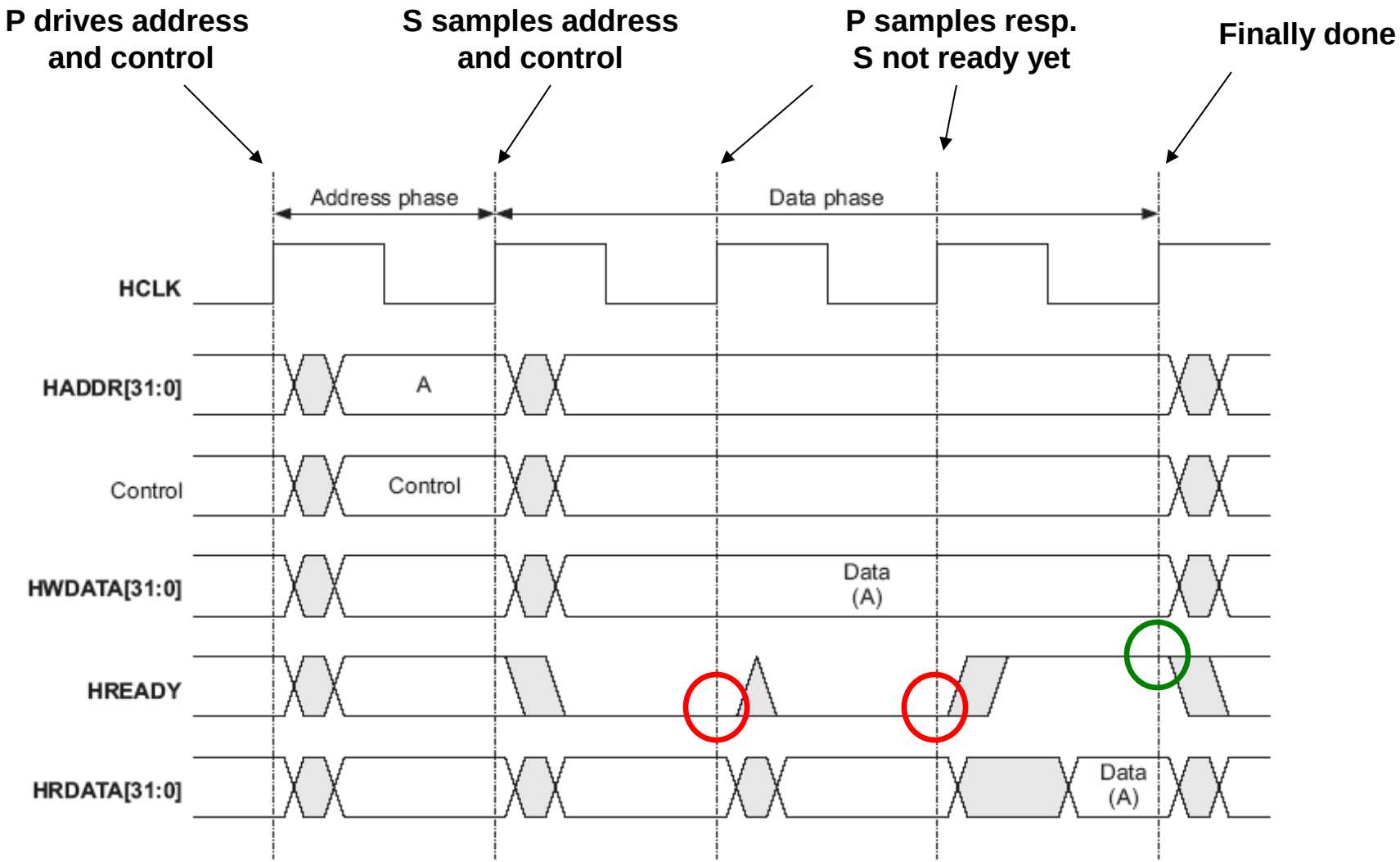
Transfer with Wait States



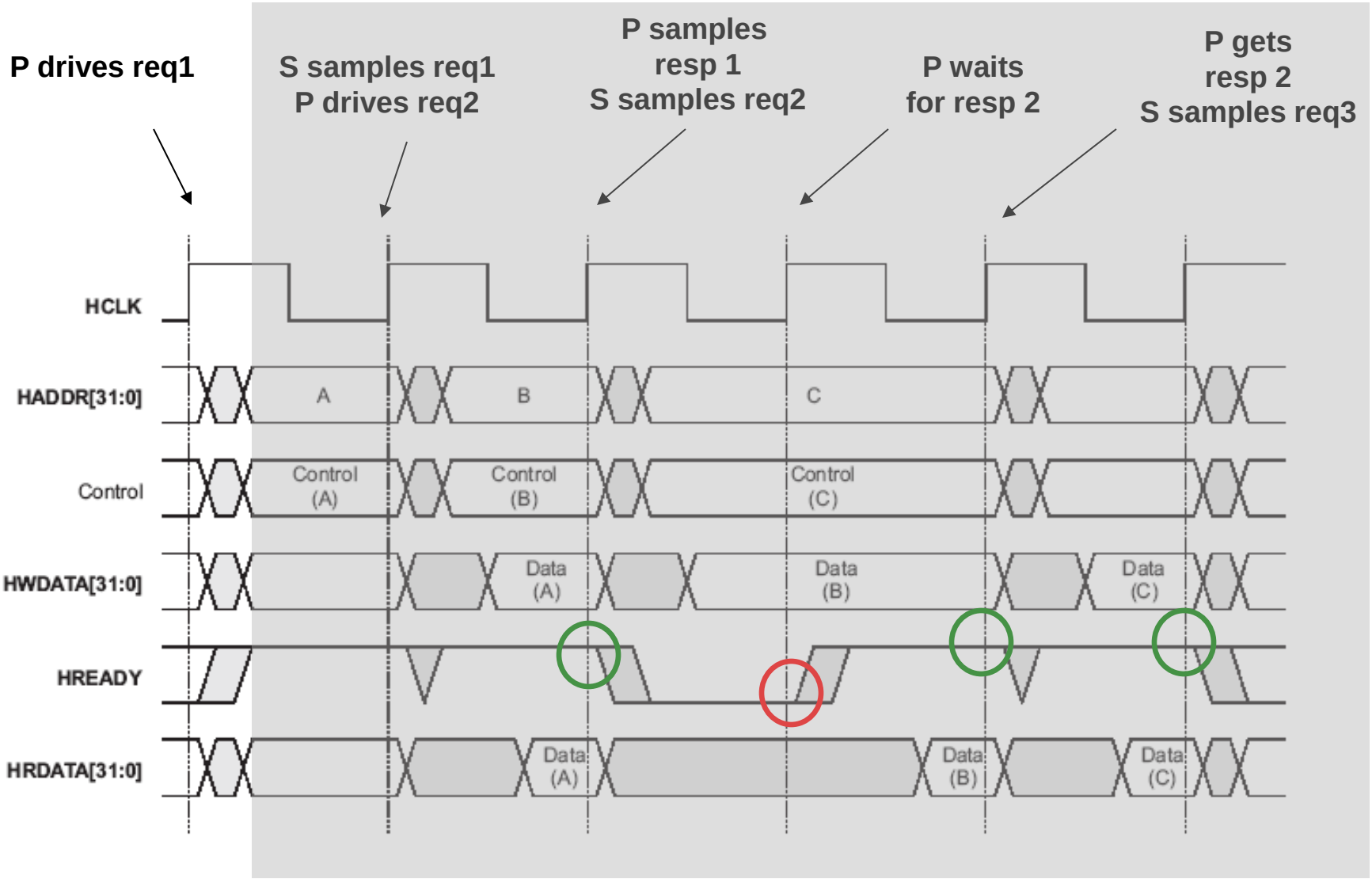
Transfer with Wait States



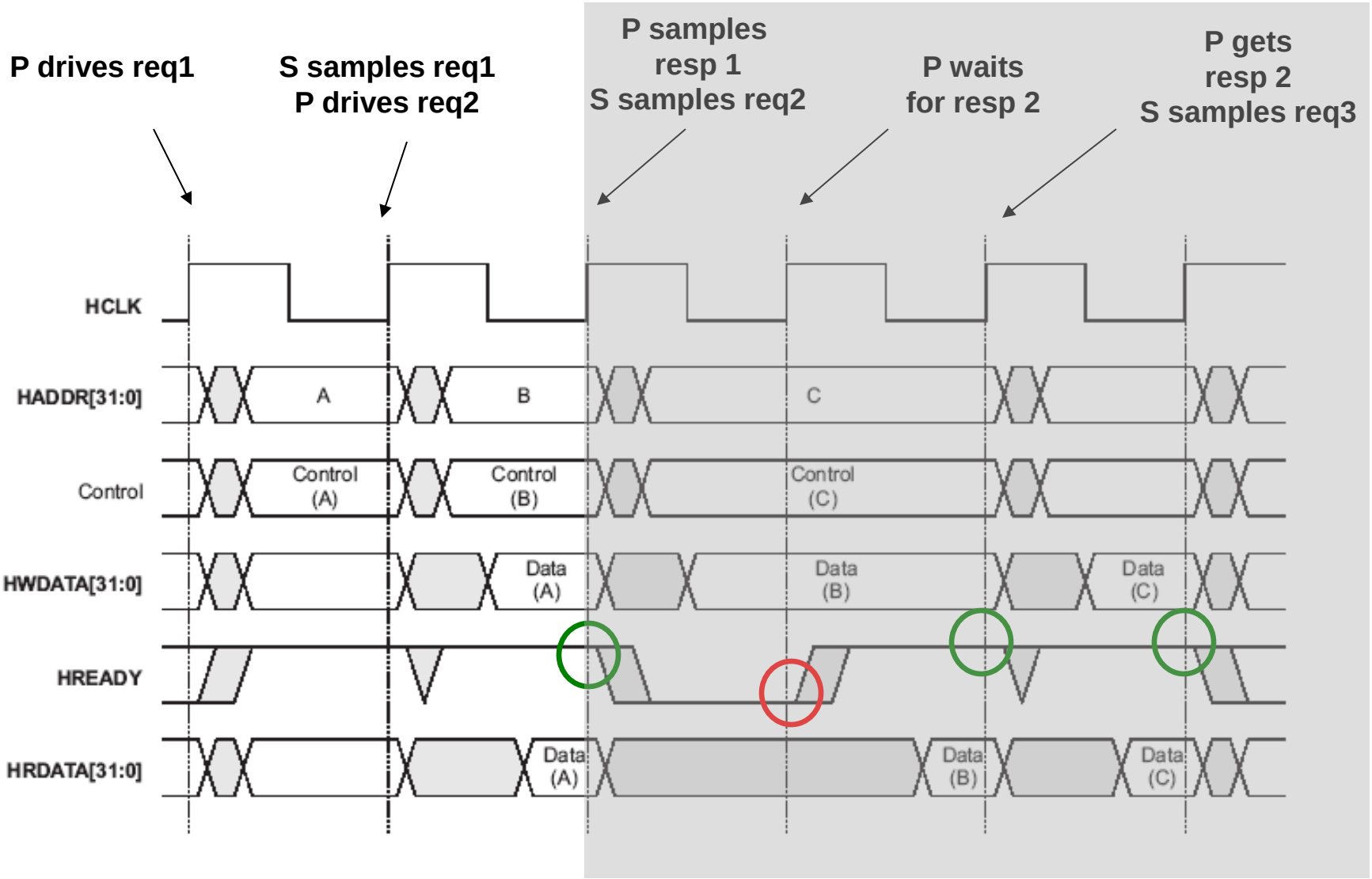
Transfer with Wait States



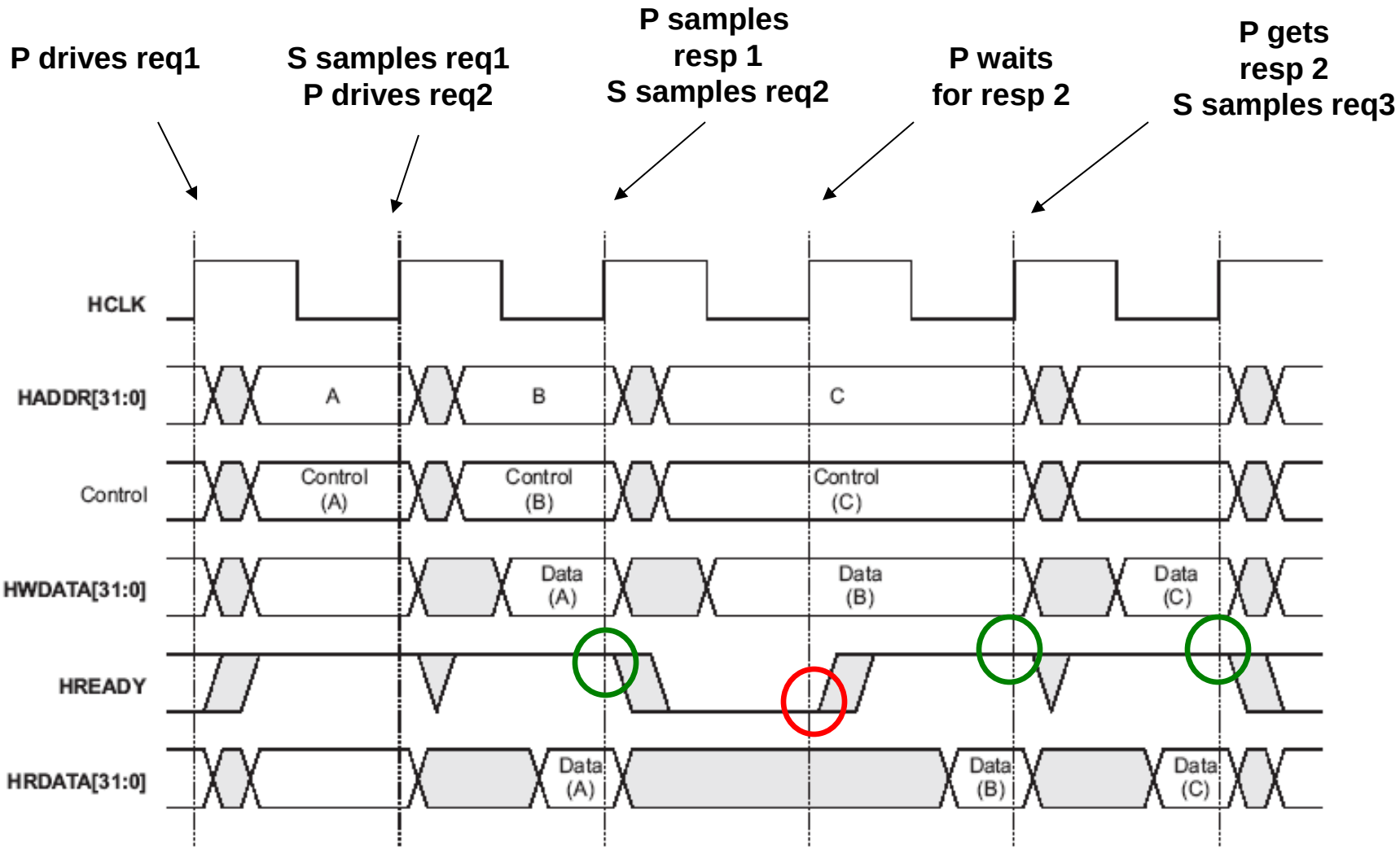
Pipelined Transfer



Pipelined Transfer



Pipelined Transfer



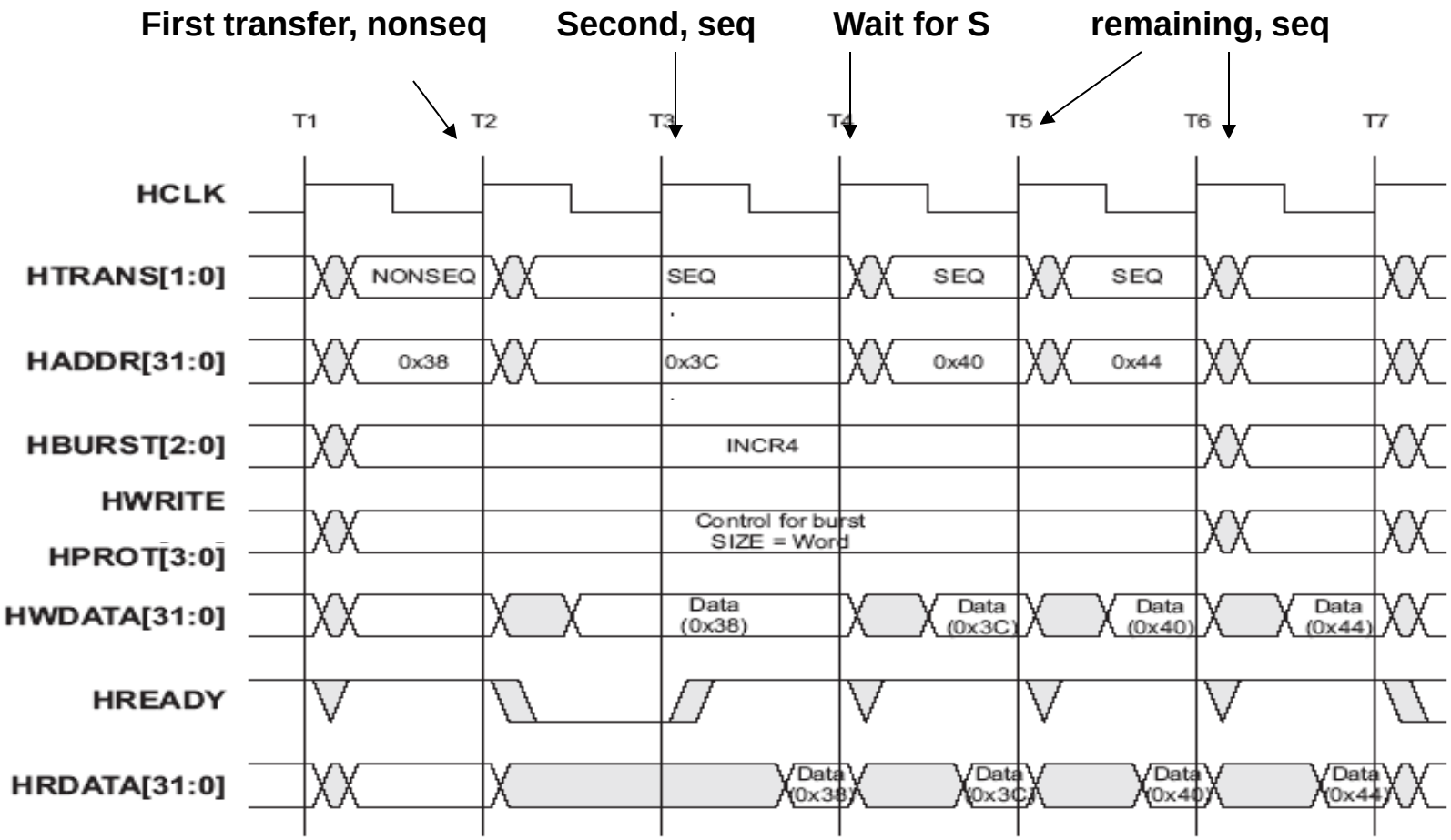
Transfer Types

- Primary drives HTRANS[1:0]
  - Idle — no data transfer
  - Busy — primary cannot prepare request in time, wait
  - NonSEQ — first transfer of a burst or single transfer
  - SEQ — remaining transfers of a burst

- Primary drives HBURST[2:0]
  - Single transfer
  - ***Burst*** transfer
    - Specified length: 4-beat, 8 beat, 16 beat
    - Unspecified length

Burst Mode

- A 4-beat increment transfer, word size



Split Transfer

- Secondary may be slow to respond
  - Waiting wastes time
  - Low bus utilization rate if many secondary units are slow
- Solution: Split a slow transfer
  - Secondary asks the primary to put the transfer on hold
    - Secondary remembers the id of the primary
  - Arbiter then grants the bus to other primary units
  - When secondary is ready, arbiter is notified
    - Secondary tells arbiter the primary id
  - Arbiter grants the bus to the primary
  - The transfer resumes
- Related signal: HRESP[1:0] from secondary

AHB summary

- AHB-lite
  - Single primary
  - No need for arbitration
  - No need for split-transfer
  - Compatible with AHB by using simple wrapper
- AHB
  - Multiple primary in the bus
  - Split transfer improves utilization rate
    - More complicated than described (think of multiple splits)

Software Issues,
Verification,
Debugging

The Ol'Reliable

Non-idealities, Optimization Approaches, and More

- ***Asynchronous*** nature of devices, events, and data
- **Data sharing** among processes
- Availability vs. lack of hardware components (e.g., FP units)

Using gprof

- Compile your code with gcc, turn on –pg flag
- Run your code, say ./a.out
  - This generates gmon.out as profiling record
- gprof ./a.out outputs profiling result

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 11.33      0.62     0.62  1770592     0.00     0.00  addlist
  9.39      1.13     0.51   611974     0.00     0.00  mrglist
  9.02      1.62     0.49   171039     0.00     0.01  getefflibs
  6.81      1.99     0.37   995358     0.00     0.00  dellist
  5.71      2.30     0.31    96195     0.00     0.01  lupdate
  4.97      2.57     0.27    96195     0.00     0.01  ldndate
  ......
```

More Accurate Tool

- **PIN** (Dynamic Binary Instrumentation Tool from Intel)
  - Free binary download
    - Has an ARM port
- **User can add** PIN tools
  - User code will be added to the original binary
- Augments binary with extra code

Optimizing Efficiency – Examples:

- **Application-Level**
- **System-Level**

Optimizing Efficiency – Examples:

- **Application-Level**
  - Loop unrolling
  - Reducing comparisons
  - Avoiding “expensive” operations (e.g., division)
  - Using better algorithms
  - Using table lookups
  - Avoiding busy waiting
  - Inlining function calls
- **System-Level**

Optimizing Efficiency – Examples:

- **Application-Level**
  - Loop unrolling
  - Reducing comparisons
  - Avoiding “expensive” operations (e.g., division)
  - Using better algorithms
  - Using table lookups
  - Avoiding busy waiting
  - Inlining function calls
- **System-Level**
  - Low-power scheduling
  - Dynamic Voltage and Frequency
  - Sleep states and DPM
    - CPU, other devices
  - Bus optimizations
  - Cache/memory optimizations
  - Use of accelerators/FPGA offload

Reading:
Power Optimization in Embedded Systems

- Papers –
  - M. Pedram – Power Optimization and Management in Embedded Systems, ASPDAC'11.
  - W. Wolf and M. Kandemir – Memory System Optimization of Embedded Software, Proc. of IEEE, Jan'03.

Debugging Techniques

- Printk, dmesg
- /proc file system
- Strace
- gdb
  - gdb vmlinux /proc/kcore
    - Kernel debugging, for example, to report the value of “jiffies”
- Simulation

GDB

- Standard debugger supporting many targets and languages
- Typical tasks
  - Setting ***break points, watch points***
  - Evaluate expressions
  - ***Trace/step a program***
  - Show/modify memory or register content
- Problem
  - GDB is too big to load into many devices
  - Solution: split into two parts

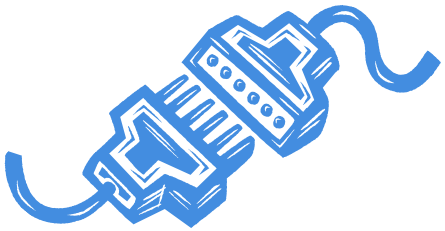
GDB Remote Debugger

- Host side: user interface
  - Loads program with symbol table
    - Can look up variable name to get memory address
  - Responds to user command
    - e.g. evaluate expression var1+b[idx],
    - e.g. run 10 instructions
  - Decompose user command to remote debugging protocol commands
- Target side: target processor interface
  - Responds to remote protocol commands
    - e.g. return memory/register content

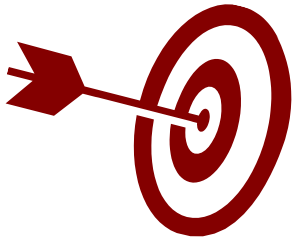
GDB Remote Debugger

- Target
  - An evaluation board with the target processor, or
  - An instruction set simulator running on host or another computer
- Deploy gdb target side
  - Gdbserver : much smaller than gdb, or
  - Link code to debug with gdb stubs which does gdbserver's task

GDB loads symbol table
Evaluates expression
Translates user command



TCP/IP
or Serial



responds to
remote protocol,
monitors execution

Typical Debugging Sequence

- Gdbserver starts, set up socket, wait
- GDB starts, connect to socket
- GDB loads binary which contains a symbol table, loads binary into gdbserver
- Users sets a break point e.g. at file1:line 100
  - Gdb looks symbol table and translates the line # to program address A
  - Gdb sends command break point A to gdbserver
  - Gdbserver replaces instruction at A with a soft interrupt
- User types “run”
  - gdb sends command to gdbserver, wait
  - Gdbserver runs and stops at break point, acknowledges gdb
  - Gdb responds to user, shows source code line

Typical Debugging Sequence

- Gdbserver starts, set up socket, wait
- GDB starts, connect to socket
- GDB loads binary which contains a symbol table, loads binary into gdbserver
- Users sets a break point e.g. at file1:line 100
  - Gdb looks symbol table and translates the line # to program address A
  - Gdb sends command break point A to gdbserver
  - Gdbserver replaces instruction at A with a soft interrupt
- User types “run”
  - gdb sends command to gdbserver, wait
  - Gdbserver runs and stops at break point, acknowledges gdb
  - Gdb responds to user, shows source code line

Typical Debugging Sequence

- Gdbserver starts, set up socket, wait
- GDB starts, connect to socket
- GDB loads binary which contains a symbol table, loads binary into gdbserver
- Users sets a break point e.g. at file1:line 100
  - Gdb looks symbol table and translates the line # to program address A
  - Gdb sends command break point A to gdbserver
  - Gdbserver replaces instruction at A with a soft interrupt
- User types “run”
  - gdb sends command to gdbserver, wait
  - Gdbserver runs and stops at break point, acknowledges gdb
  - Gdb responds to user, shows source code line

Typical Debugging Sequence

- Gdbserver starts, set up socket, wait
- GDB starts, connect to socket
- GDB loads binary which contains a symbol table, loads binary into gdbserver
- Users sets a break point e.g. at file1:line 100
  - Gdb looks symbol table and translates the line # to program address A
  - Gdb sends command break point A to gdbserver
  - Gdbserver replaces instruction at A with a soft interrupt
- User types “run”
  - gdb sends command to gdbserver, wait
  - Gdbserver runs and stops at break point, acknowledges gdb
  - Gdb responds to user, shows source code line

Debugging

A program, simpleIO.c, is given to you. Compile the code using:

```
gcc -ggdb -fno-stack-protector -o simpleIO simpleIO.c
```

Run the code. You'll see that the code may give segmentation fault depending on the input string size. Without changing simpleIO.c at all, can you make the code print out "Hacked!!! This function was not supposed to run!" before the segmentation fault occurs?

Hints:

1. gdb hints:

- You can use `disass` command in gdb to see the assembly code of different functions and investigate how much memory has been allocated for the buffers, local variables, etc.

- "`gdb -q simpleIO`" will let you dig into more details. You can set a breakpoint at main(). Run your binary with gdb, and proceed step by step after the breakpoint. When prompted, enter a string made of "A"s. Once you see the "puts(buff)" command displayed, print out rsp and rbp values by using "x/x $rsp" and "x/x $rbp" commands in gdb. Continue printing $rbp-4, $rbp-8, … using x/x until you see a data value of 0x…4141. Hexadecimal code of A is 41, so type "x/s <address where you saw 0x…4141>" to confirm the value is indeed the same as the one you entered. What does this analysis tell you about the stack frame organization and about the space allocated for the buffer?

- Can you figure out where the return address of the GetInput() function is located in the stack? The code will use this return address to continue after GetInput() function finishes execution.

2. You can pass an input argument into an executable by piping the input string. E.g., printf "AAAA" | ./simpleIO is the same as typing ./simpleIO and then entering AAAA into the command line prompt.

3. You can place hexadecimal numbers into a string in a Linux terminal window as follows:

```
printf '\xaa\xab\xac\n'
printf '\x30\x31\x32\n'
printf "AAAA\xFF\x00"  (this one appends AAAA with some hex numbers)
```

Submission

- Write down your command (that achieves the task above) for running simpleIO in answers.txt.
- Explain briefly why your input argument causes the NeverExecutes() function to run and how you came up with the command line input / which steps you followed in gdb. No need to submit any code.
- Please submit on GradeScope. Make sure to include all team members' names in the txt file.

Names/BU usernames of team members:

Command:
preparing_my_arguments_in_one_line_if_needed
I_can_pipe_something_here ./simpleIO I_can_also_add_something_here

Explanation:
This code works because my inputs are awesome. This part should be around 600 characters (not strict, but please don't write an essay).

Steps:

Explain the steps you have followed to find the command argument in another 600 characters

- Using this tool, I found something in the code
- The code normally works this way.
- By doing _____, I can trick the program so it executes NeverExecutes().