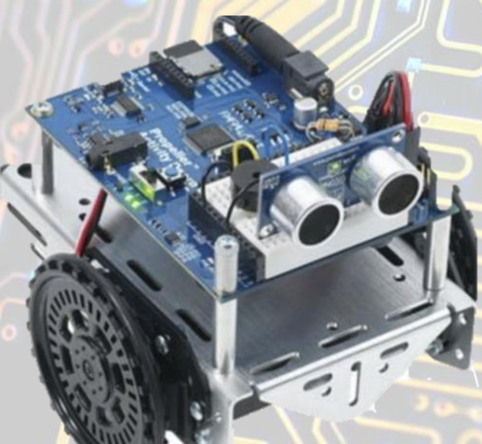
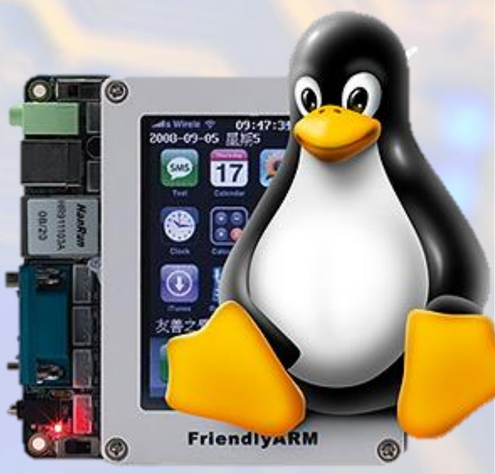
EC535 Introduction to Embedded Systems

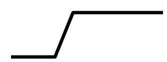




Timing Diagrams

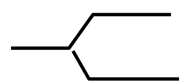
- Most common method for describing a communication protocol
- Time proceeds to the right on x-axis

- Control signal: low or high

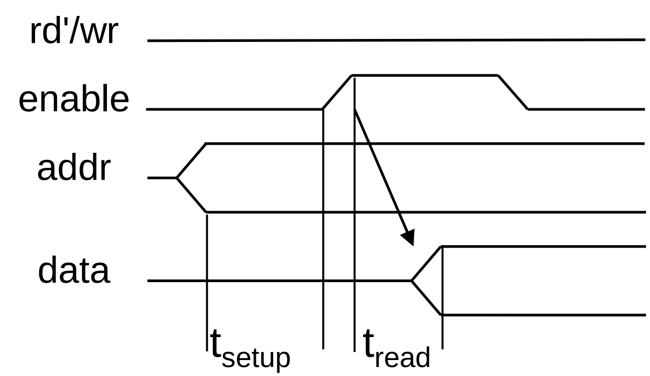


  - May be active low (e.g., go’, /go, or go_L)
  - Use terms *assert* (active) and *deassert*
  - Asserting go’ means go=0

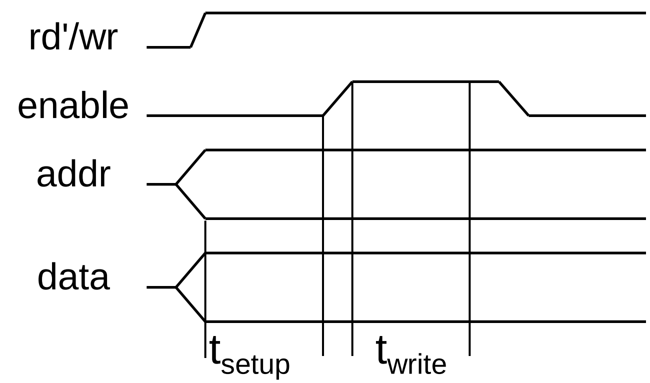
- Data signal: not valid or valid



- Protocol may have subprotocols
  - Called bus cycle, e.g., read and write

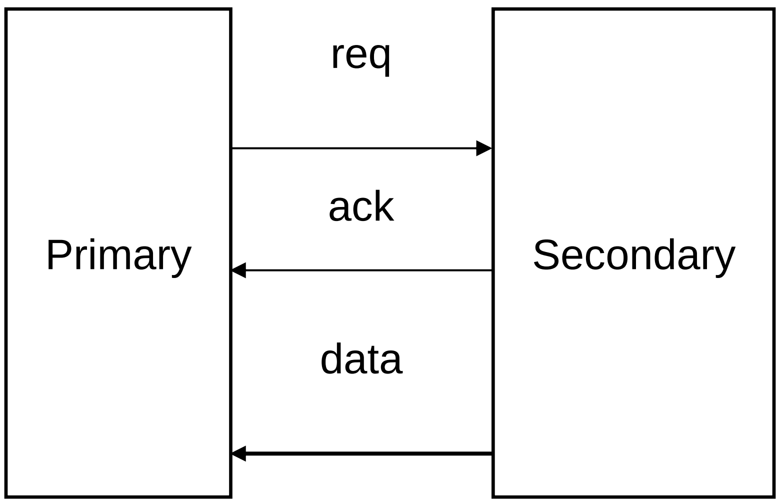


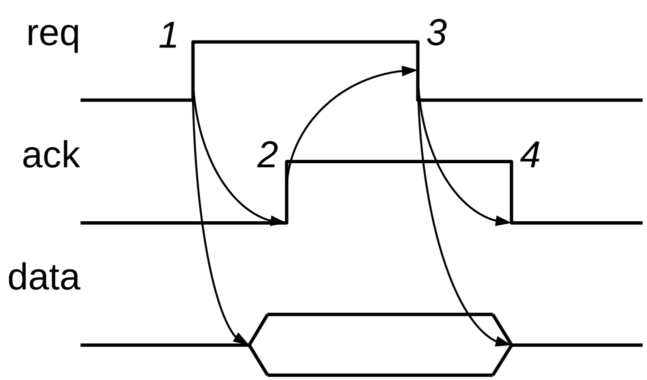
example read protocol



example write protocol

Basic protocol concepts: control methods





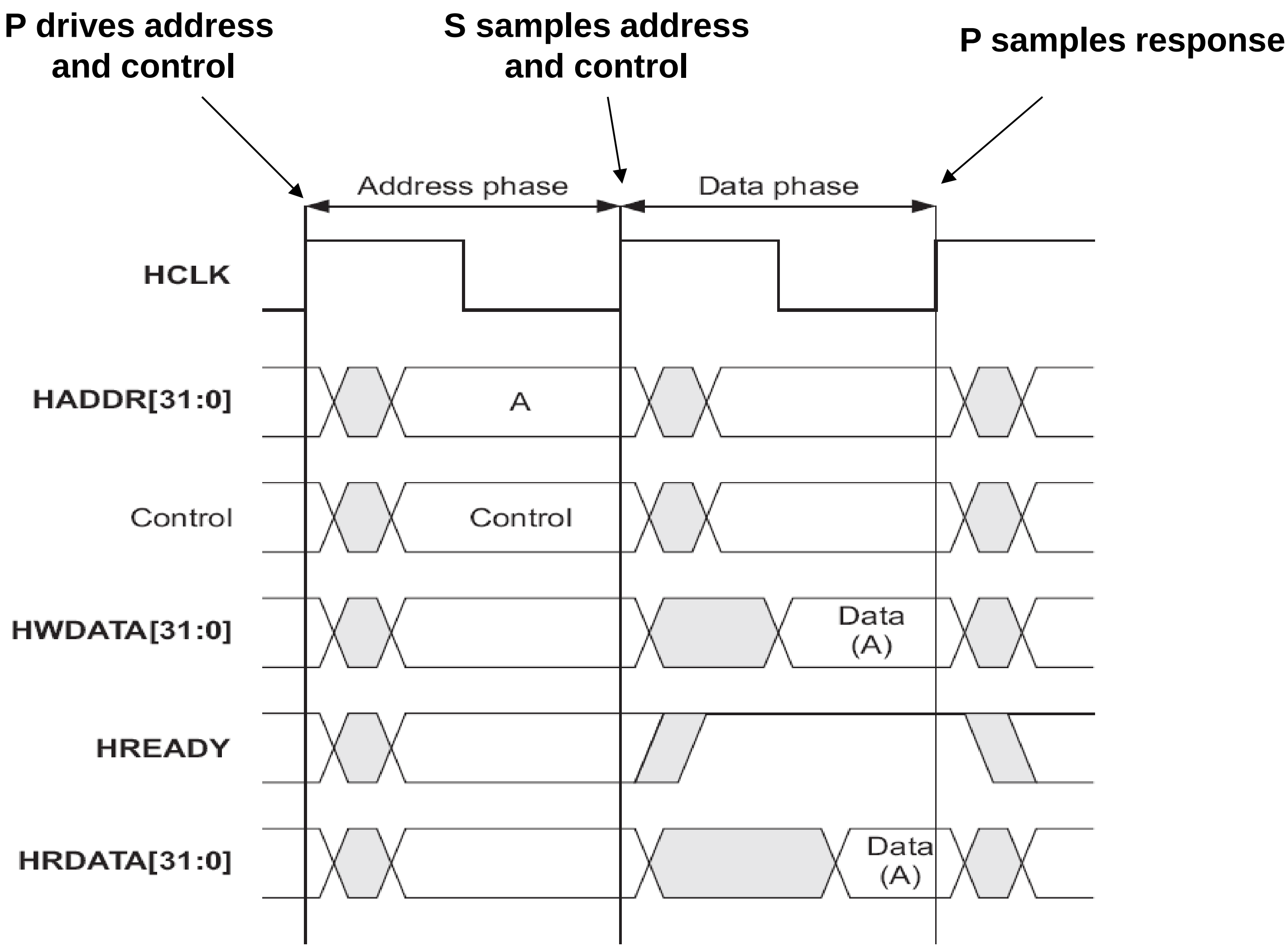
1. Primary asserts *req* to receive data

2. Secondary puts data on bus
and asserts *ack*

3. Primary receives data and deasserts *req*
4. Secondary ready for next request

Handshake protocol

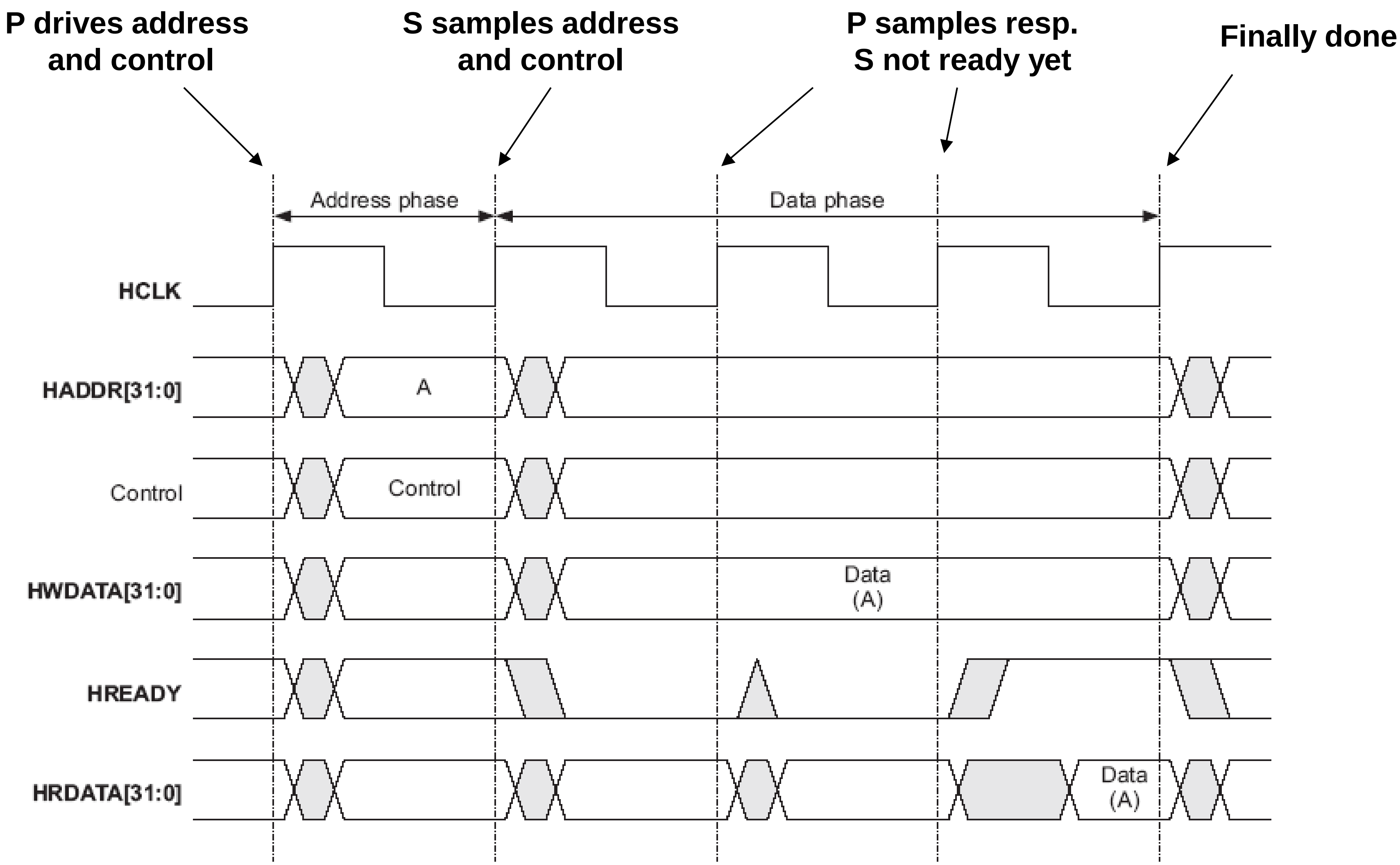
A Simple Transfer



AHB Main Signals

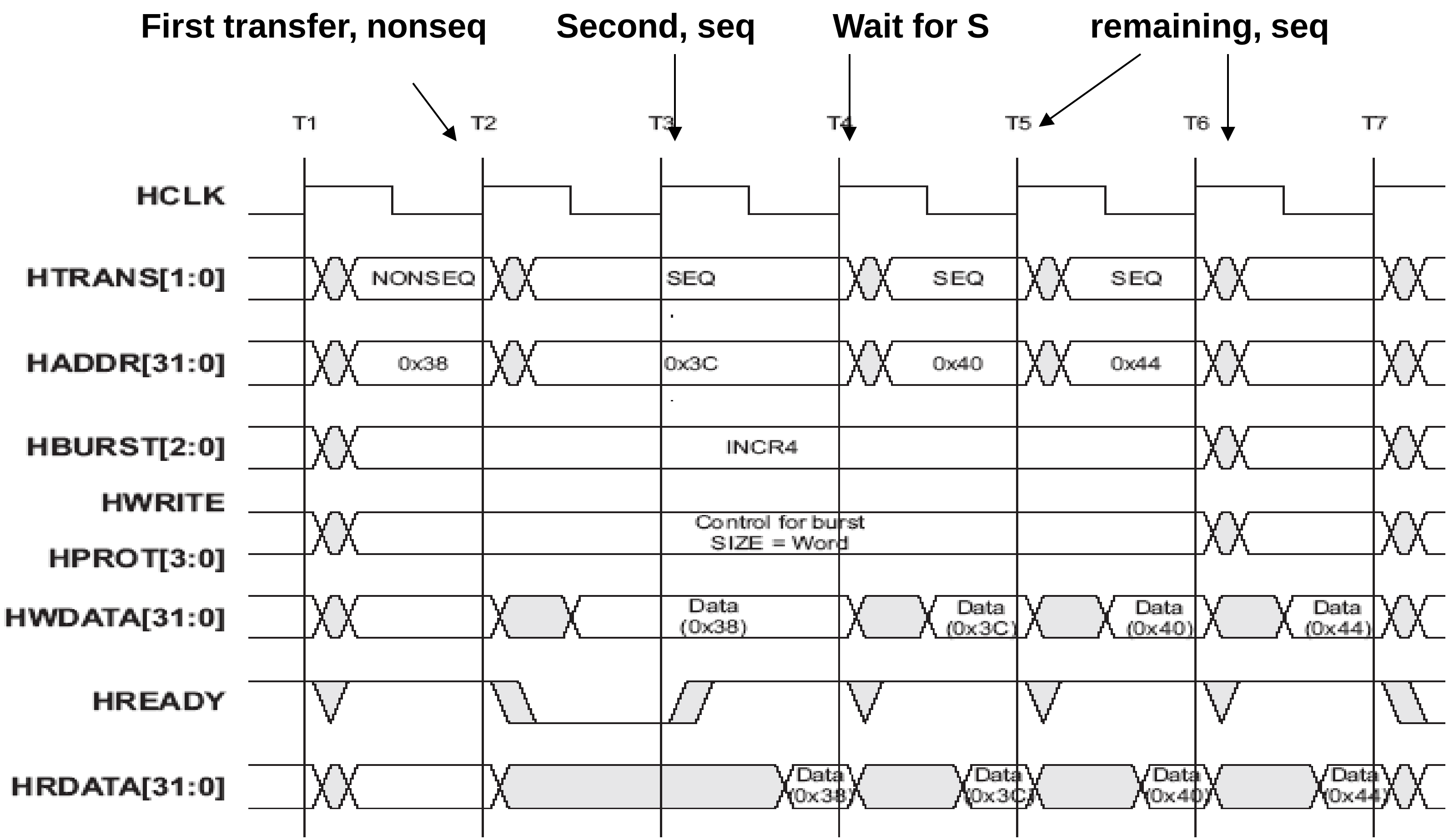
name	source	
HCLK	Clock source	Global clock for all bus transactions
HADDR[31:0]	Primary	32bit address line
HWRITE	P	HI — write, LOW — read
HSIZE[2:0]	P	Transfer size (8bits – 1024bits)
HWDATA[31:0]	P	Data to write
HSELx	Decoder	Selects intended secondary
HREADY	Secondary	HI — a transfer finishes, LOW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

Transfer with Wait States



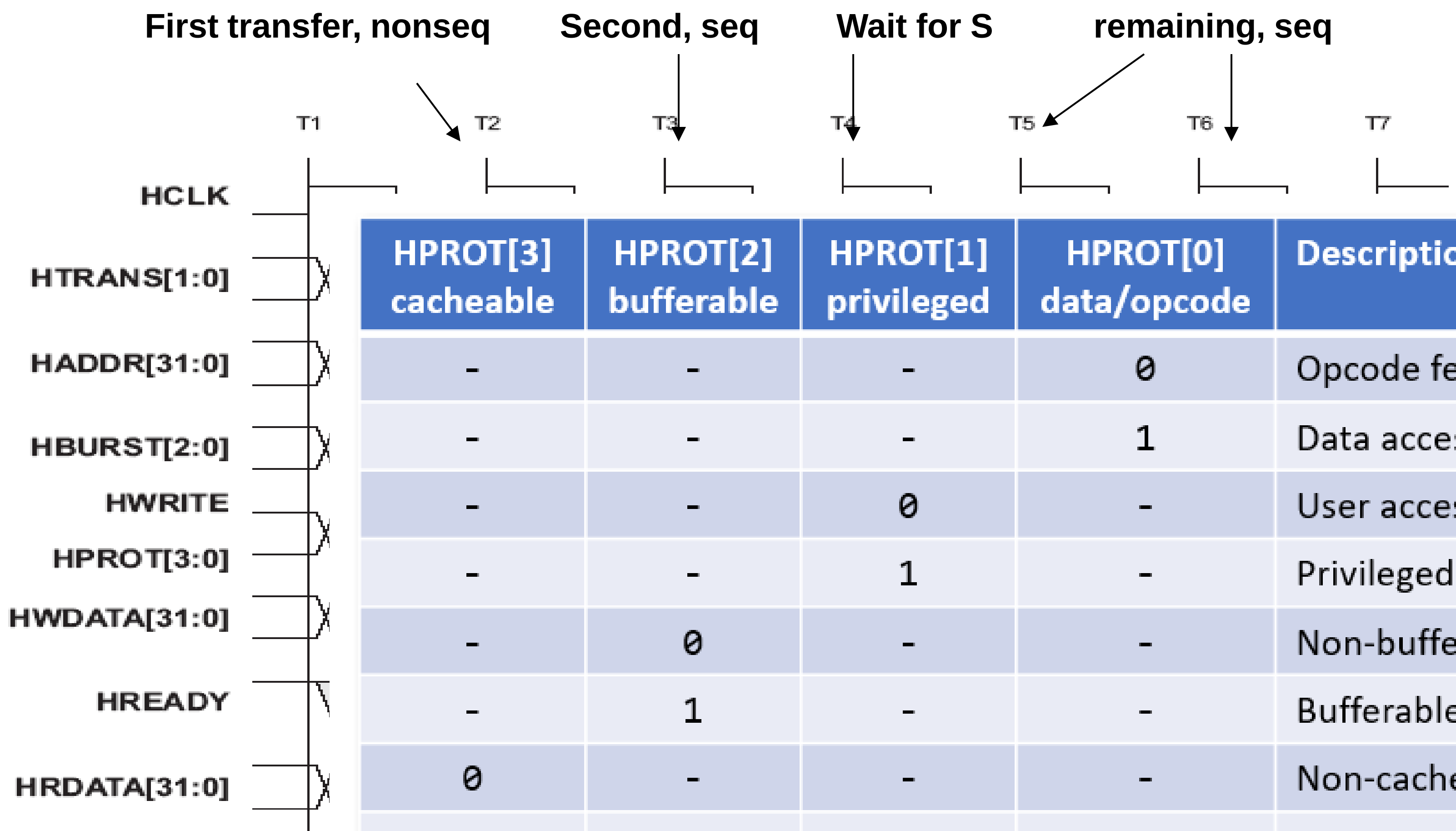
Burst Mode

- A 4-beat increment transfer, word size



Burst Mode

- A 4-beat increment transfer, word size



HPROT[3] cacheable	HPROT[2] bufferable	HPROT[1] privileged	HPROT[0] data/opcode	Description
-	-	-	0	Opcode fetch
-	-	-	1	Data access
-	-	0	-	User access
-	-	1	-	Privileged access
-	0	-	-	Non-bufferable
-	1	-	-	Bufferable
0	-	-	-	Non-cacheable
1	-	-	-	Cacheable

https://ww1.microchip.com/downloads/en/devicedoc/39582b.pdf

REGISTER 9-3: SSPSTAT: MSSP STATUS REGISTER (I^2C MODE) (ADDRESS 94h)

R/W-0	R/W-0	R-0	R-0	R-0	R-0	R-0	R-0
SMP	CKE	D/$\overline{A}$	P	S	R/$\overline{W}$	UA	BF
bit 7							bit 0

bit 7 **SMP:** Slew Rate Control bit

In Master or Slave mode:
1 = Slew rate control disabled for standard speed mode (100 kHz and 1 MHz)
0 = Slew rate control enabled for high-speed mode (400 kHz)

bit 6 **CKE:** SMBus Select bit

In Master or Slave mode:
1 = Enable SMBus specific inputs
0 = Disable SMBus specific inputs

bit 5 **D/$\overline{A}$:** Data/$\overline{\text{Address}}$ bit

In Master mode:
Reserved.

In Slave mode:
1 = Indicates that the last byte received or transmitted was data
0 = Indicates that the last byte received or transmitted was address

bit 4 **P:** Stop bit

1 = Indicates that a Stop bit has been detected last
0 = Stop bit was not detected last

Note: This bit is cleared on Reset and when SSPEN is cleared.

bit 3 **S:** Start bit

1 = Indicates that a Start bit has been detected last
0 = Start bit was not detected last

Note: This bit is cleared on Reset and when SSPEN is cleared.

bit 2 **R/$\overline{W}$:** Read/$\overline{\text{Write}}$ bit information (I^2C mode only)

In Slave mode:
1 = Read
0 = Write

Note: This bit holds the R/$\overline{W}$ bit information following the last address match. This bit is only valid from the address match to the next Start bit, Stop bit or not $\overline{\text{ACK}}$ bit.

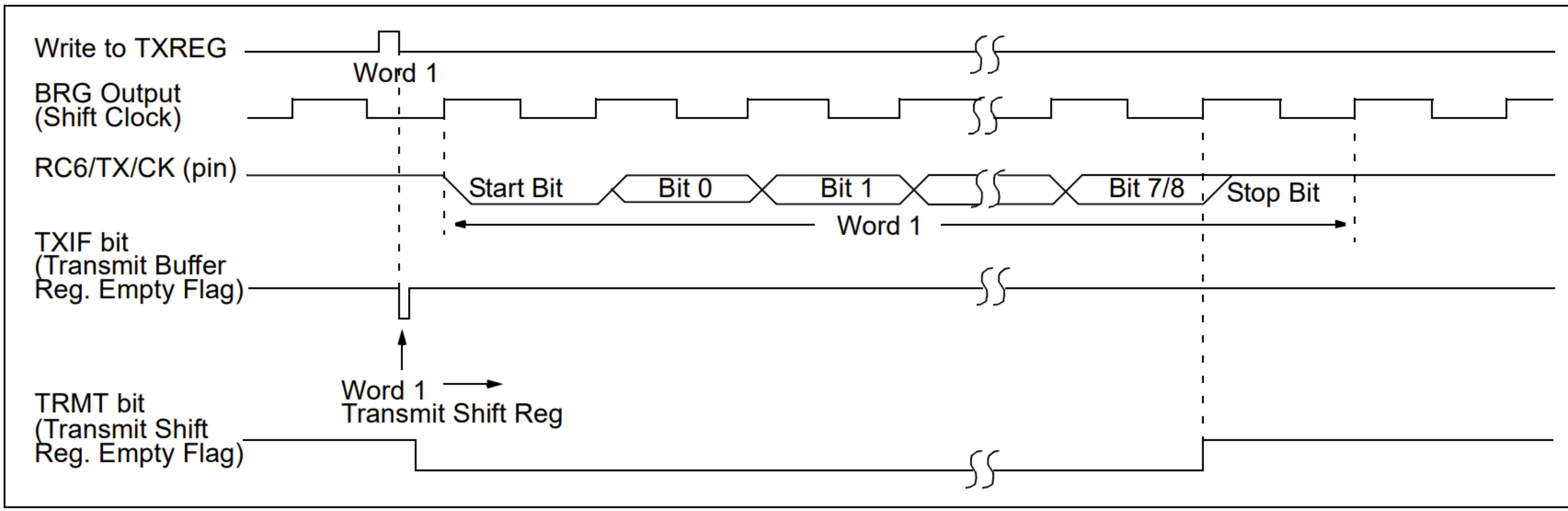
In Master mode:
1 = Transmit is in progress
0 = Transmit is not in progress

Note: ORing this bit with SEN, RSEN, PEN, RCEN or ACKEN will indicate if the MSSP is in Idle mode.

bit 1 **UA:** Update Address (10-bit Slave mode only)

1 = Indicates that the user needs to update the address in the SSPADD register
0 = Address does not need to be updated

FIGURE 10-2: ASYNCHRONOUS MASTER TRANSMISSION

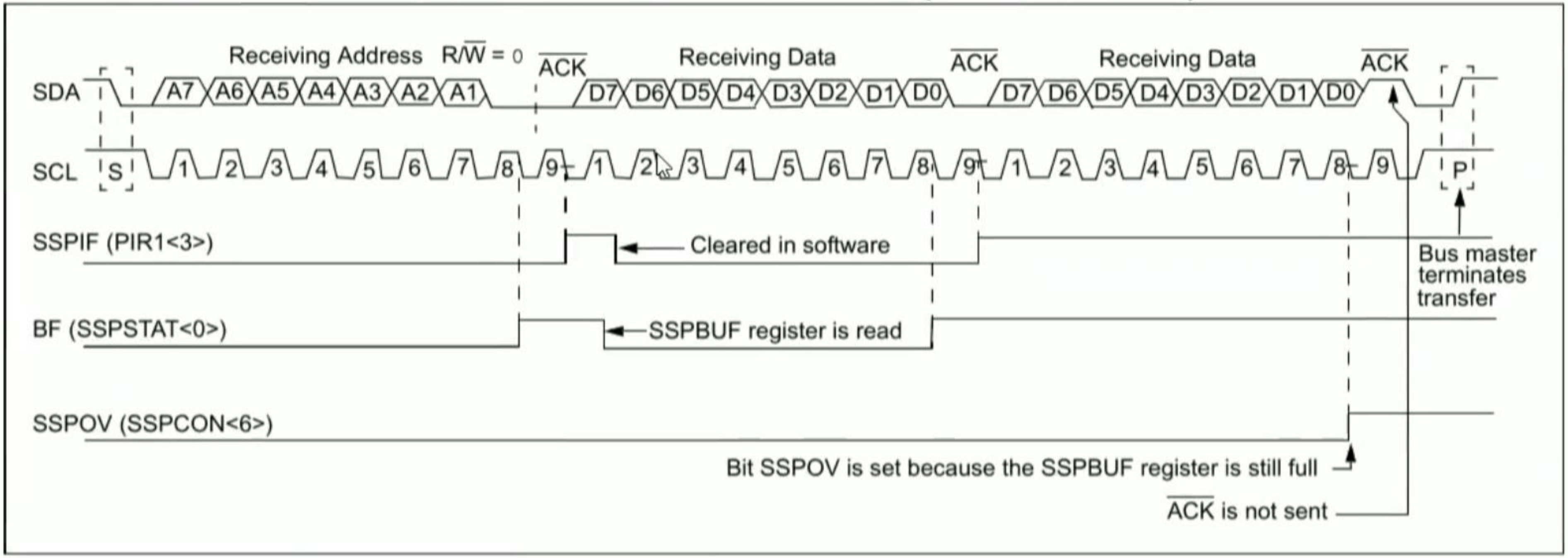


https://ww1.microchip.com/downloads/en/devicedoc/39582b.pdf

REGISTER 9-3: SSPSTAT: MSSP STATUS REGISTER (I²C MODE) (ADDRESS 94h)

R/W-0	R/W-0	R-0	R-0	R-0	R-0	R-0	R-0
SMP	CKE	D/$\overline{A}$	P	S	R/$\overline{W}$	UA	BF
bit 7							bit 0

FIGURE 10-6: I²C™ WAVEFORMS FOR RECEPTION (7-BIT ADDRESS)



Note: ORing this bit with SEN, RSEN, PEN, RCEN or ACKEN will indicate if the MSSP is in Idle mode.

bit 1 **UA:** Update Address (10-bit Slave mode only)
1 = Indicates that the user needs to update the address in the SSPADD register
0 = Address does not need to be updated

FIGURE 9-4: SLAVE SYNCHRONIZATION WAVEFORM

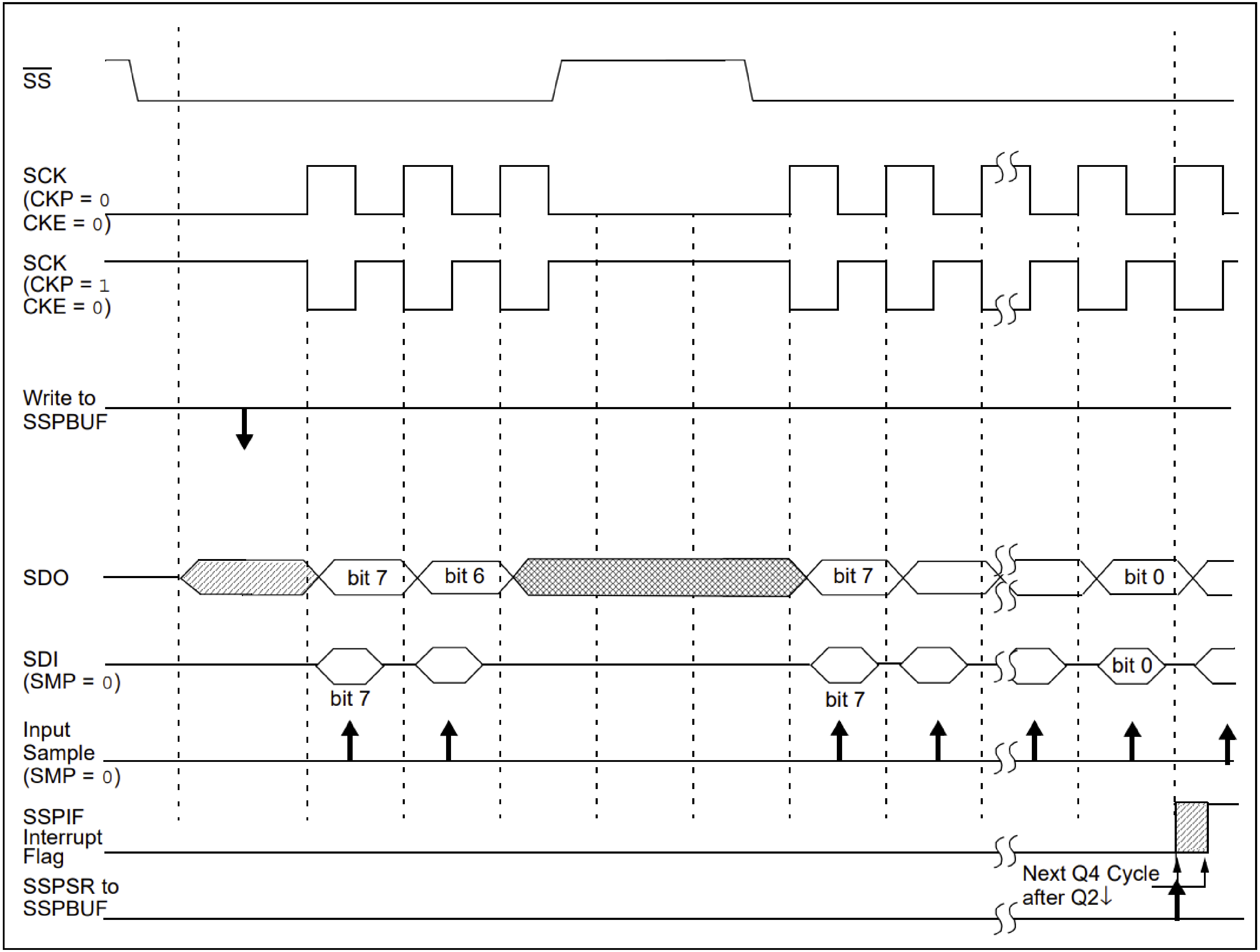
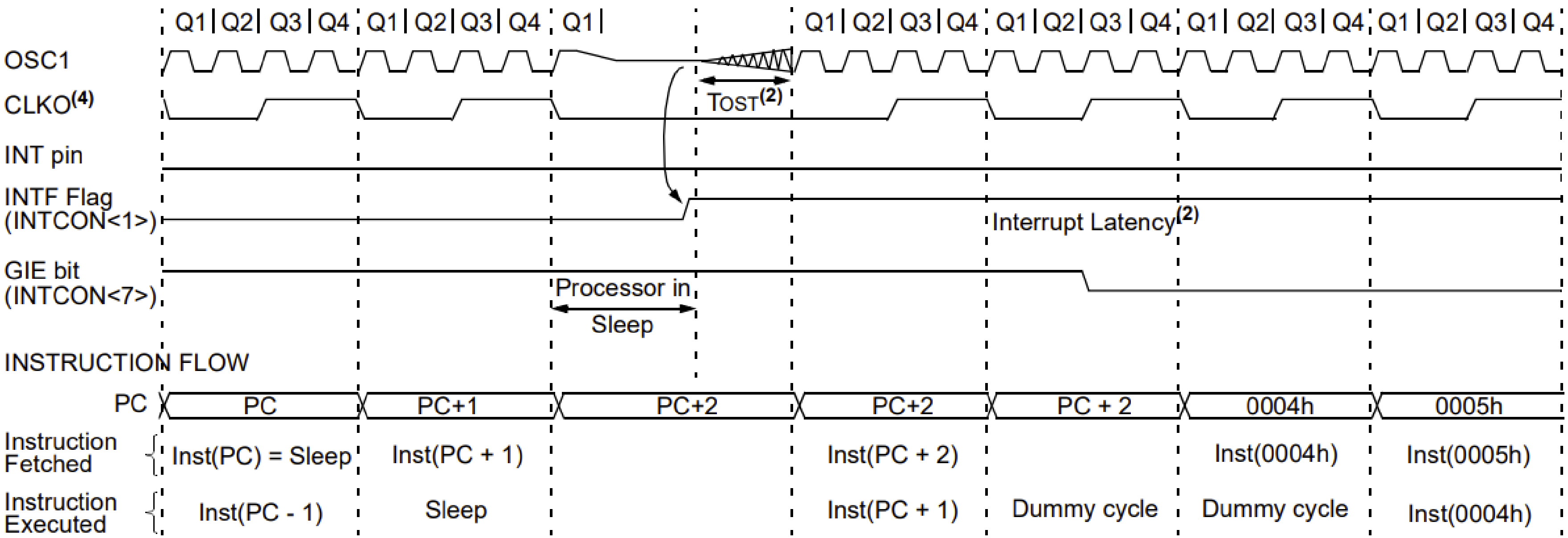


FIGURE 14-12: WAKE-UP FROM SLEEP THROUGH INTERRUPT



Note 1: XT, HS or LP Oscillator mode assumed.

2: TOST = 1024 TOSC (drawing not to scale). This delay will not be there for RC Oscillator mode.

3: GIE = `1` assumed. In this case, after wake- up, the processor jumps to the interrupt routine. If GIE = `0`, execution will continue in-line.

4: CLKO is not available in these oscillator modes but shown here for timing reference.

Software Issues,
Verification,
Debugging

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

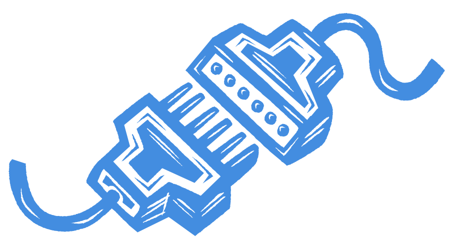
GDB Remote Debugger

- 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

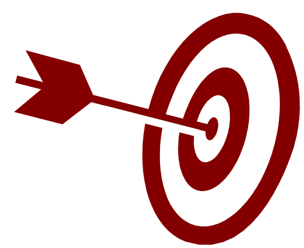
GDB Remote Debugger

- 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.

Command:
printf "AAAAAAAAAAAAAAAAAAAAAAAA\x54\x05\x40\x00" |
./simpleIO

Explanation:
Address of NeverExecutes() in the binary has the addres 0x400554. By writing the address in the input string, I can overwrite the GetInput()'s return address so that it now goes to NeverExecutes() instead of returning back to main().

Steps: - gdb -q simpleIO- r- Enter a long String (e.g.,27 Bytes): 'AAAAAAAAAAAAAAAAAAAAAAAAAAA'- Gives a segmentation fault- bt- I observe that the last 3 bytes on the stack get overwritten on with AAA because of the buffer overflow in GetInput(). This causes the segmentation fault. - quit and run gdb again.
- disass NeverExecutes() : gives the address of this function as - 0x00400554- run the executable as printf "AAAAAAAAAAAAAAAAAAAAAAAA\x54\x05\x40\x00" | ./simpleIO-

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().

Exam: *4/15*

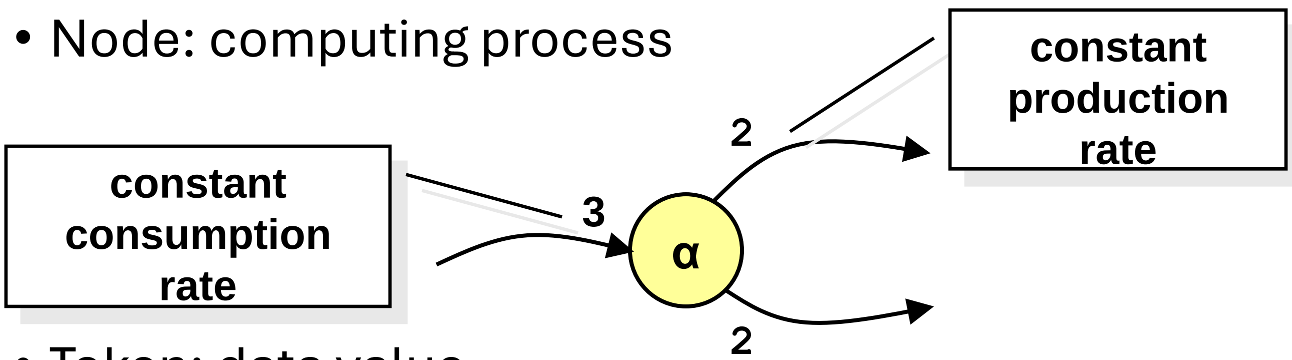
- One page of notes
  - **No internet or phone**
- Questions
  - Multiple-choice
  - True-false with brief explanation
  - Problem solving (including light calculation)
- Coverage: comprehensive
  - Topics from lectures, including in-class exercises
  - HWs, labs (but no programming task during exam)
  - Reading material (excluding manuals and datasheets)

Exam

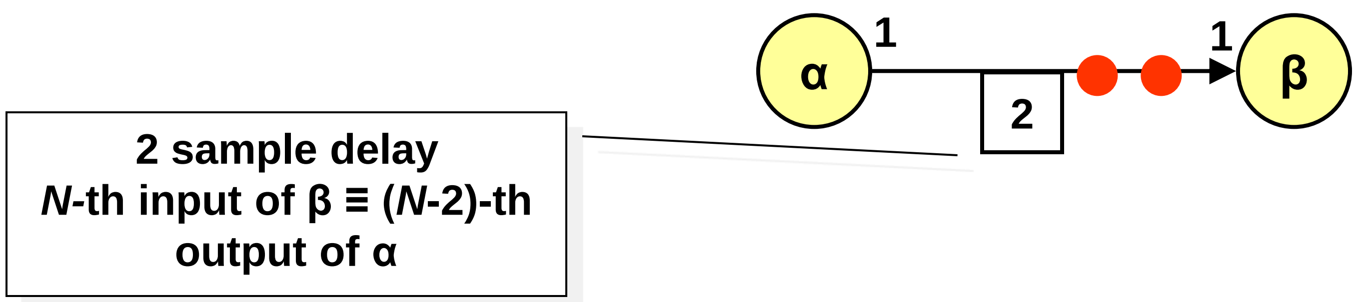
- SDF properties, SDF scheduling
- RTOS concepts, scheduling algorithms, inter-process communication
- Linux scheduler
- Bus, arbitration
- Memory technologies
- ISA, ARM properties
- ASIPs
- Lab tasks
    - QEMU, kernel image, rootfs
    - Device driver concepts
    - User program – device driver interactions

SDF — Definitions

- Node: computing process

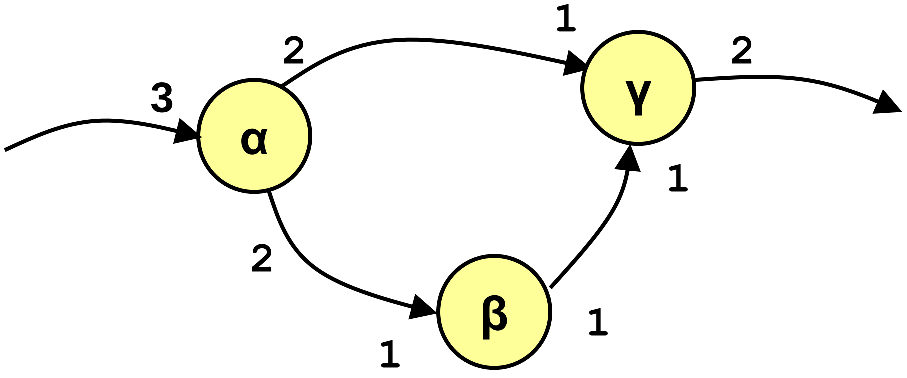


- Token: data value
- Arc: contains a fifo
- Delay: fifo initialized with # token(s)



SDF

- A node can execute when enough tokens are available on all of the inputs.
- Always produces and consumes the same fixed number of tokens for each invocation.
  - The flow of data does not depend on values of the data.



SDF Scheduling

- Solve linear equations to determine the period
  - Inconsistent system has only all-zero solution
- Simulate SDF execution to find a PASS
  - List scheduling
    - Repeat: find a node that can execute
      - If its execution count equal to prescribed by step1, skip
      - Execute the node
  - On success, should return to initial state after 1 period
  - Otherwise, there must be a deadlock
- SDF Implementation in hardware

RTOS Concepts

- Process: unique execution of a program.
- Thread = lightweight process
- Task: A process or a thread.
- Context/Context switching
- Kernel
- Priority: static, dynamic
- Scheduler:
  - Non-preemptive/cooperative
  - Preemptive

RTOS

- General knowledge of Linux kernel
  - Scheduling states
  - Scheduling policy, priority levels
  - What is a preemptive Linux kernel (or kernel preemption)?
  - What is a jiffy?
- Interprocess communication
  - Critical section
  - Mutual exclusion schemes

Linux Device Drivers

- Layers of software
  - Device driver, VFS, system call, library, user program
  - Difference between read system call and file_ops->read
- Kernel module
- Device file
  - File operation via file_ops
- Printk, procfs
- Kernel timer
- Memory copying

Communication Architecture

- Topologies
  - Bus, crossbar, p2p…
- Bus concepts
  - Primary-secondary, arbiter, decoder
  - Transmitter (sender), receiver
  - Arbitration schemes
  - Serial protocols
    - I2C, CAN
  - Parallel protocols
    - Pipelining, burst-mode, split-transfer
    - AMBA

ISA

- The aspects of a computer architecture visible to a programmer
  - Instruction syntax, semantics, binary encoding
  - Registers and memory model, data types
  - Interrupt, exception, external I/O
- Longer lifetime than particular implementation
  - Reuse of software
  - Sometimes hampers new innovations
- ARM ISA characteristics
  - Register file
  - Instruction types, specialties

Basic Computer Organization Concepts

- CISC, RISC
- Endianness
- Pipelining
- Hazards
  - Data
  - Control
  - Structural
- Hazard resolution techniques

- Von Neumann vs. Harvard

Energy Efficiency

- Power vs energy management
- Performance and energy costs of energy management
  - Sleep / wakeup delays, breakeven time
- Policies
  - Timeout, advantages / limitations
- Scheduling for low power
- Dynamic voltage and frequency scaling
  - Implementation, policies, etc.
- Thermal challenges and reliability

Simulation and Software

- Types of simulators
  - Instruction set simulator
    - Interpreter: easy to use as debugger (e.g., ema)
    - Static compiled simulator
    - Dynamic compiled simulator: Qemu
  - Microarchitecture simulator
    - Sima, gem5, SimpleScalar, etc.
    - Slow, but provides more feedback (cycle #, cache stats, etc.)
- Debugging methods
  - gdb
  - Kernel debugging
  - Others? (JTAG, ICE, etc.)