Please note that Cypress is an Infineon Technologies Company.
The document following this cover page is marked as “Cypress” document as this is the company that originally developed the product. Please note that Infineon will continue to offer the product to new and existing customers as part of the Infineon product portfolio.

Continuity of document content
The fact that Infineon offers the following product as part of the Infineon product portfolio does not lead to any changes to this document. Future revisions will occur when appropriate, and any changes will be set out on the document history page.

Continuity of ordering part numbers
Infineon continues to support existing part numbers. Please continue to use the ordering part numbers listed in the datasheet for ordering.

PSoC® 1 Fast Unsigned Multiplication Algorithms

Author: Dave Van Ess
Associated Project: Yes
Associated Part Family: All PSoC® 1 Devices
Software Version: PSoC Designer™ 5.4
Related Application Notes: None

AN2032 details efficient high-speed algorithms for multiplication using the PSoC® 1's built-in Multiply-accumulate (MAC) unit. Topics include signed versus unsigned values, the basics of the PSoC 1 MAC, using macros, and multi-byte multiplication.

Introduction

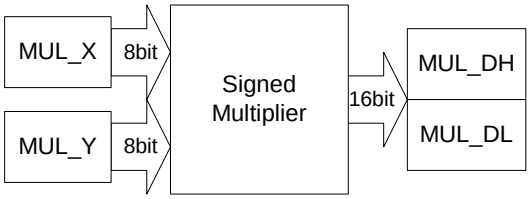
Performing multiplication can be tedious without dedicated hardware. Any dedicated hardware multiplier has finite resolution and applications may arise where more resolution is required. This application note describes the techniques that enable the PSoC 1 MCU to quickly multiply multiple-byte-operands and give a full resolution output.

This application uses the PSoC 1 MAC (Multiply/Accumulate) resource.

The PSoC 1 Multiplier

Figure 1 shows that four registers are used to access the multiplier.

Figure 1. PSoC 1 Multiplier Block Diagram



These four registers are all located in register Bank 0, having the following names and addresses:

MUL_X	(Bank 0 E8h)
MUL_Y	(Bank 0 E9h)
MUL_DH	(Bank 0 EAh)
MUL_DL	(Bank 0 EBh)

MUL_X and MUL_Y are "write only" registers where signed 2's-complement 8-bit values are input to the multiplier. The signed, 2's-complement 16-bit output is deposited into two 8-bit "read only" registers, MUL_DH (upper) and MUL_DL (lower). The following program code is a simple example to multiply the contents of XVal (-1) and YVal (-1) and deposit the answer in "Result."

Code 1

```
;-----------------------------------------
;  This program places the contents of
;  "XVal"and "Yval" into the multiplier.
;  The answer is then moved to "Result"
;-----------------------------------------
export _main
include "m8c.inc"
export Result
export XVal
export YVal
area bss(RAM)
   Result:    BLK  2   ; 16 bit
   XVal:      BLK  1   ; 8 bit
   YVal:      BLK  1   ; 8 bit
area text(ROM,REL)
_main:
   mov [XVal],-1       ; initialize data
   mov [YVal],-1

   mov A,[XVal]
   mov reg[MUL_X],A
   mov A,[YVal]
   mov reg[MUL_Y],A
   mov A,reg[MUL_DL]
   mov [Result + 1],A
   mov A,reg[MUL_DH]
   mov [Result],A
   loop:               ; done!
   jmp loop
```

Note The result is stored in big-endian format. The high byte of Result is stored in [Result + 0] and the low byte is stored in [Result + 1]

When Code 1 runs on the ICE and halts at the end of the calculation, the contents of RAM show that "Result" is 1.

Macros

There are a lot of examples where data is pushed into and retrieved from the multiplier. The following macros eliminate the repetitive tedium. All these macros are located in *unsignedmath.inc*, located in the project file associated with this application note.

Macro to fill MUL_X:

```
macro PushMulX
;  reg[MUL_X] = @0
   mov A,[@0]
   mov reg[MUL_X],A
endm
```

Macro to fill MUL_Y:

```
macro PushMulY
;  reg[MUL_Y] = @0
   mov A,[@0]
   mov reg[MUL_Y],A
endm
```

Macro to retrieve the lower byte of the answer:

```
macro GetLSB
   mov A,reg[MUL_DL]
endm
Macro to retrieve the upper byte:
macro GetSignedMSB
   mov A,reg[MUL_DH]
endm
```

Macro that ties it all together:

```
macro Multiply16s_8s_8s
;  @0 = @1 * @2
;  multiplies two 8-bit, signed values and
returns
;  a 16-bit, signed value.
   PushMulX @1
   PushMulY @2
   GetLSB
   mov [@0 + 1],A
   GetSignedMSB
   mov [@0],A
endm
```

Code 1 program code example has been rewritten to use these new macros and is shown in Code 2:

Code 2

```
;-----------------------------------------
;  This program places the contents of
;  "XVal"and "Yval" into the multiplier.
;  The answer is then moved to "Result"
;-----------------------------------------
export _main
include "m8c.inc"
include "unsignedmath.inc"
export Result
export XVal
export YVal
area bss(RAM)
   Result:     BLK  2    ; 16 bit
   XVal:       BLK  1    ; 8 bit
   YVal:       BLK  1    ; 8 bit
area text(ROM,REL)
_main:
   mov [XVal],-1         ; initialize data
   mov [YVal],-1

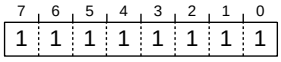
   Multiply16s_8s_8s Result, XVal, YVal
   loop:                 ; done!
   jmp loop
```

It takes 42 CPU cycles to retrieve data, perform the multiply operation, and place the answer in memory. For a CPU speed of 24 MHz, this works out to 1.75 µs. Signed 2's-complement multiplies can be done quickly.

Signed and Unsigned Values

This section explains the difference between a signed value and an unsigned value.

Figure 2. A Byte Filled With Ones



The value of the byte in Figure 2 depends on how the byte's contents are interpreted. The on-board MAC interprets its inputs as signed, 8-bit integers. Performing unsigned multiplication requires extra work.

Equation 1 defines the relationship between unsigned and signed values:

$$x_{unsigned} = x_{signed} + 256 f(x) \quad f(x) = \begin{cases} 1 & x_{bit7} = 1 \\ 0 & x_{bit7} = 0 \end{cases}$$ Equation 1

Applying Equation 1 to the byte contents in Figure 2 shows that it is valid.

$$255_{unsigned} = -1_{signed} + 256 \quad f(-1_{signed})$$

Multiplying two single, unsigned operands produces an unsigned result:

$$R_{unsigned} = x_{unsigned} y_{unsigned}$$ Equation 2

Using Equation 1 to substitute for the unsigned terms in Equation 2, results in Equation 3:

$$R_{unsigned} = (x_{signed} + 256 f(x))(y_{signed} + 256 f(y))$$ Equation 3

Expanding Equation 3 produces a clearer Equation 4:

$$R_{unsigned} = \begin{matrix} 65536 f(x) f(y) + \\ 256 f(x) y_{signed} + \\ 256 f(y) x_{signed} + \\ x_{signed} y_{signed} \end{matrix}$$ Equation 4

As stated in Equation 1, multiplication of two unsigned bytes produces an unsigned 2-byte result. Parsing Equation 4 yields the following results:

[$x_{signed}y_{signed}$] is the 2-byte output of the signed multiplier.

[256*f*(y)x_{signed}] has the effect of adding x_{signed} to the upper byte of the result if y_{bit7} is 1.

[256*f*(x)y_{signed}] has the effect of adding y_{signed} to the upper byte of the result if x_{bit7} is 1.

[65536*f*(x)*f*(y)] has a value of either zero or it falls outside the range of the result; it can be ignored.

Equation 5 puts this together to show how the resultant upper byte and lower byte for an unsigned multiply are calculated:

$$\begin{aligned} (R_{unsigned})_H &= MUL_DH + f(x) y_{signed} + f(y) x_{signed} \\ (R_{unsigned})_L &= MUL_DL \end{aligned}$$ Equation 5

Note The lower byte result of an unsigned multiply is equal to the lower byte result of a signed multiply. If the results of any multiplication are truncated to have the same resolution as the inputs, there is no difference between signed and unsigned multiplication.

The following segment code calculates the unsigned MSB byte of an unsigned multiply:

Code 3

```
;-------------------------------------
;
;  Code to get an unsigned MBS in A
;
;-------------------------------------
mov A,reg[MUL_DH]       ; get the data
tst [XVal],80h
jz skip_y
   add A,[YVal]
skip_y:
tst [YVal],80h
jz skip_x
   add A,[XVal]
skip_x:
```

The following macro performs the same function:

```
macro GetUnsignedMSB
; @0 @1 & MUL_DH determine value
   mov A,reg[MUL_DH]
   tst [@0],80h
   jz .+4 ;(skip_y)
      add A,[@1]
;  skip_y
   tst [@1],80
   jz .+4 ;(skip_x)
      add A,[@0]
;  skip_x
endm
```

The following macro multiplies unsigned bytes and stores the resultant answer in memory:

```
macro Multiply16u_8u_8u
;  @0 = @1 * @2
;   multiplies two 8-bit, unsigned values
and returns
;  a 16-bit, unsigned value.
   PushMulX @1
   PushMulY @2
   GetLSB
   mov [@0 + 1],A
   GetUnsignedMSB @1, @2
   mov [@0],A
endm
```

Code 4 is similar to Code 3, except that this time the unsigned resultant is stored:

Code 4

```
;-----------------------------------------
;  This program places the contents of
;  "XVal"and "Yval" into the multiplier.
;  The unsigned answer is then moved to
;  "Result"
;-----------------------------------------
export _main
include "m8c.inc"
include "unsignedmath.inc"
export Result
export XVal
export YVal
area bss(RAM)
   Result:    BLK  2   ; 16 bit
   XVal:      BLK  1   ; 8 bit
   YVal:      BLK  1   ; 8 bit
area text(ROM,REL)
_main:
   mov [XVal],ffh      ; initialize data
   mov [YVal],ffh

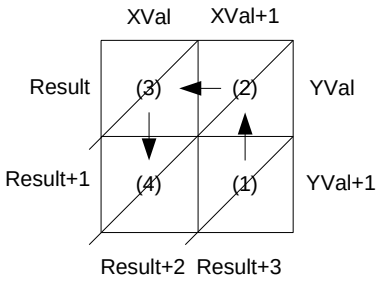
   Multiply16u_8u_8u Result, XVal, YVal
   loop:               ; done!
      jmp loop
```

When this program is run on the ICE and halted at the end of calculation, the contents of RAM show that the "Result" is 255*255 or FE01h. The time it takes to perform the retrieval of data, the actual multiply, and placement of the unsigned answer in memory is 80 CPU cycles. For a CPU speed of 24 MHz this works out to 3.33 usec.

Multiple Byte Multiplication

If two unsigned, 2-byte operands are multiplied together, the result is an unsigned, 4-byte answer. Figure 3 shows that this multiplication is the combination of four smaller 8-bit, unsigned multiplications.

Figure 3. Napier Matrix for 16-Bit Multiply



The following four-step algorithm illustrates four, smaller 8-bit unsigned multiplications:

1. Unsigned multiply of XVal+1 and YVal+1 with the lower byte resultant stored in Result+2 and the upper byte stored in Result+3.
2. Move up and multiply XVal+1 and YVal and place the upper resultant byte in Result+1, and add the lower resultant byte with Result+2. By moving only one space horizontally each time, only one multiplicand needs to be reloaded, cutting the overhead of loading the multiplier nearly in half.
3. Move left, multiply XVal and YVal, store the upper resultant byte in Result, and add the lower resultant byte with Result+1.
4. Move down, multiply XVal and YVAL+1, add the upper resultant byte to Result+1, and then add the lower resultant byte to Result+2.

The following macro implements this algorithm:

```
macro Multiply32u_16u_16u
;  result = XVal * YVal
;  @0 = @1 * @1
;  16bit by 16 bit unsigned multiply
;  with 32 bit unsigned result
;
;  (1)
   Multiply16u_8u_8u  (@0  +  2),(@1  +1),(@2
+1)
;  (2)
   PushMulY  @2
   GetUnsignedMSB  (@1 +1),@2
   mov [@0 +1],A
   GetLSB
   add [@0 + 2],A
   adc [@0 + 1],0
;  (3)
   PushMulX @1
   GetUnsignedMSB @1,@2
   mov [@0],A
   GetLSB
   add [@0 +1 ],A
   adc [@0],0
;  (4)
   PushMulY (@2 + 1)
   GetUnsignedMSB, @1,(@2 + 1)
   push A
   GetLSB
   add [@0 + 2],A
   pop A
   adc [@0 +1],A
   adc [@0],0
endm
```

Code 5 is similar to Code 4, but this time 16-bit, unsigned values are multiplied and the stored unsigned resultant is 32 bits:

Code 5

```
;-----------------------------------------
;   This program places the contents of
"XVal"
;   and "Yval" into the multiplier. The
unsigned
;  answer is then moved to "Result"
;-----------------------------------------
export _main
include "m8c.inc"
include "unsignedmath.inc"

export Result
export XVal
export YVal

area bss(RAM)
   Result:    BLK  4   ; 32 bit
   XVal:      BLK  2   ; 16 bit
   YVal:      BLK  2   ; 16 bit
area text(ROM,REL)
_main:
   mov [XVal],-1       ; initialize data
   mov [XVal+1],-1
   mov [YVal],-1

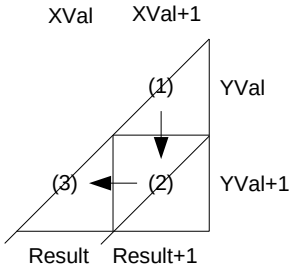
   Multiply32u_16u_16u Result, XVal, YVal
   loop:               ; done!
      jmp loop
```

When this program is run on the ICE and halted at the end of the calculation, the contents of RAM show that the "Result" is FFFE0001h. The time it takes to perform the retrieval of data, the actual multiply, and placement of the unsigned answer in memory is 333 CPU cycles. For a CPU speed of 24 MHz this works out to 13.8 usec.

Truncated 16-bit - A Valid Subset

If the results of any multiplication are truncated to have the same resolution as the inputs, there is no difference between signed and unsigned multiplication. Often, the result of a 16-bit multiply is truncated to just 16 bits. In this case, unsigned calculations yield valid results. Figure 4 shows a subset of the matrix in Figure 3.

Figure 4. Matrix for 16-Bit Multiply, Truncated Result



Because only the lower two bytes of the result are needed, less than half of the full calculation is required. It does not matter if the inputs are signed or unsigned.

This macro implements the Napier Matrix in Figure 4:

```
macro Multiply16_16_16
;  result = XVal * YVal
;  @0 = @1 * @1
;  16 bit by 16-bit multiply
;  with a 16-bit result

;  (1)x1y0  r1
   PushMulY  (@2 +0)
   PushMulX  (@1 +1)
   GetLSB
   mov [@0],A
;  (2)x1y1 r01
   PushMulY (@2 +1)
   GetLSB
   mov [@0 +1],A
   GetUnsignedMSB  (@1 +1), (@2 +1)
   add [@0 +0],A
;  (3)x0y1 r1
   PushMulX (@1 +0)
   GetLSB
   add [@0 +0],A
endm
```

Code 6 is similar to Code 5, but this time 16-bit values are multiplied and the stored resultant is 16 bits:

Code 6

```
;-----------------------------------------
;   This program places the contents of
"XVal"
;   and "Yval" into the multiplier. The
unsigned
;  answer is then moved to "Result"
;-----------------------------------------
export _main
include "m8c.inc"
include "unsignedmath.inc"
export Result
export XVal
export YVal
area bss(RAM)
   Result:    BLK  2   ; 32 bit
   XVal:      BLK  2   ; 16 bit
   YVal:      BLK  2   ; 16 bit
area text(ROM,REL)
_main:
   mov [XVal],-1       ; initialize data
   mov [XVal+1],-1
   mov [YVal],-1

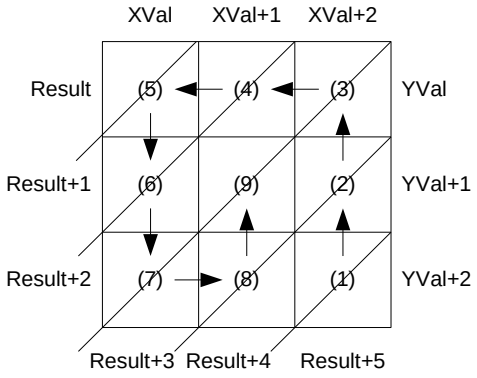
   Multiply16_16_16 Result, XVal, YVal
   loop:               ; done!
      jmp loop
```

When Code 6 is run on the ICE and halted at the end of the calculation, the contents of RAM show that the "Result" is 1. The time it takes to perform the retrieval of data, the actual multiply, and placement of the unsigned answer in memory is 115 CPU cycles. For a CPU speed of 24 MHz this works out to 4.8 usec.

24-Bit Operands 48-Bit Result

Floating-point numbers store their mantissa as a 24-bit, unsigned value. A 24-bit multiplier goes most of the way to making a floating-point multiplication routine. You only deal with the sign, exponent adjustment, zero, infinity, and NAN. Figure 5 shows the Napier Matrix for a 24-bit, unsigned multiply with a 48-bit, unsigned result.

Figure 5. Napier Matrix for 24-Bit Multiply



Following is macro code example for 24-bit, unsigned multiply with a 48-bit, unsigned result:

```
macro Multiply48u_24u_24u
;  result = Xval * Yval
;    @0 = @1 * @2
;   24 bit by 24-bit, unsigned multiply
;   with a 48-bit, unsigned result
;

;  (1) x2 y2 r45
   Multiply16u_8u_8u  (@0  +  4),(@1  +2),(@2
+2)
;  (2) x2 y1 r34
   PushMulY  (@2 +1)
   GetUnsignedMSB  (@1 +2), (@2 +1)
   mov [@0 +3],A
   GetLSB
   add [@0 + 4],A
   adc [@0 + 3],0
;  (3) x2y0 r23
   PushMulY  (@2 +0)
   GetUnsignedMSB  (@1 +2), (@2 +0)
   mov [@0 +2],A
   GetLSB
   add [@0 + 3],A
   adc [@0 + 2],0
;  (4) x1y0 r12
   PushMulX  (@1 +1)
   GetUnsignedMSB  (@1 +1), (@2 +0)
   mov [@0 +1],A
   GetLSB
   add [@0 + 2],A
   adc [@0 + 1],0
;  (5) x0y0 r01
   PushMulX  (@1 +0)
   GetUnsignedMSB  (@1 +0), (@2 +0)
   mov [@0 +0],A
   GetLSB
   add [@0 + 1],A
   adc [@0 + 0],0
;  (6) x0y1 r012
   PushMulY  (@2 +1)
   GetUnsignedMSB  (@1 +0), (@2 +1)
   push A
   GetLSB
   add [@0 + 2],A
   pop A
   adc [@0 + 1],A
   adc [@0 + 0],0
;  (7) x0y2 r0123
   PushMulY  (@2 +2)
   GetUnsignedMSB  (@1 +0), (@2 +2)
   push A
   GetLSB
   add [@0 + 3],A
   pop A
   adc [@0 + 2],A
   adc [@0 + 1],0
   adc [@0 + 0],0
;  (8) x1y2 r01234
   PushMulX  (@1 +1)
   GetUnsignedMSB  (@1 +1), (@2 +2)
   push A
   GetLSB
   add [@0 + 4],A
   pop A
   adc [@0 + 3],A
   mov A,0
   adc [@0 + 2],A
   adc [@0 + 1],A
   adc [@0 + 0],A
;  (9) x1y1 r0123
   PushMulY  (@2 +1)
   GetUnsignedMSB  (@1 +1), (@2 +1)
   push A
   GetLSB
   add [@0 + 3],A
   pop A
   adc [@0 + 2],A
   adc [@0 + 1],0
   adc [@0 + 0],0
endm
```

The time this macro takes to fetch the contents of RAM, perform a 24-bit unsigned multiply and return the 48-bit unsigned result back to RAM is 802 CPU cycles. For a CPU speed of 24 MHz this works out to 33.4 usec. The rest of the overhead needed to perform a floating-point multiply could be done in 15 usec. This makes for a floating-point multiply that takes less than 50 usec.

Summary

An understanding of the differences between signed and unsigned values makes fast, multi-byte multiplication possible. Table 1 gives a summary of the different multiplication routines discussed in this application note.

The ideas discussed here enable users to develop their own multiplication routines customized for their particular data processing requirements.

Table 1. Multiplication Routine Summary

Operands	Result	CPU Cycles	Size in Bytes
8 Bit	8 Bit	31	12
8-Bit Signed	16-Bit Signed	42	16
8-Bit Unsigned	16-Bit Unsigned	80	28
16 Bit	16 Bit	115	46
16-Bit Unsigned	32-Bit Unsigned	333	89
24-Bit Unsigned	48-Bit Unsigned	802	279

Document History

Document Title: PSoC® 1 Fast Unsigned Multiplication Algorithms – AN2032

Document Number: 001-40408

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	1532004	FKL	10/02/07	New publication of existing application note
*A	2648131	DWV	01/27/09	Included all PSoC devices
*B	3232551	GIR	04/18/11	Updated Title, Template and Abstract
*C	3276072	GIR	06/07/11	No Change.
*D	3755037	ARVI	09/25/2012	Minor updates to document. Updated template.
*E	4455578	PMAD	07/24/2014	Updated to PSoC Designer 5.4

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Automotive	cypress.com/go/automotive
Clocks & Buffers	cypress.com/go/clocks
Interface	cypress.com/go/interface
Lighting & Power Control	cypress.com/go/powerpsoc cypress.com/go/plc
Memory	cypress.com/go/memory
Optical Navigation Sensors	cypress.com/go/ons
PSoC	cypress.com/go/psoc
Touch Sensing	cypress.com/go/touch
USB Controllers	cypress.com/go/usb
Wireless/RF	cypress.com/go/wireless

PSoC® Solutions

psoc.cypress.com/solutions

PSoC 1 | PSoC 3 | PSoC 5

Cypress Developer Community

Community | Forums | Blogs | Video | Training

Technical Support

cypress.com/go/support

PSoC is a registered trademark of Cypress Semiconductor Corp. "Programmable System-on-Chip, and " PSoC Designer are trademarks of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are the property of their respective owners.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

Phone : 408-943-2600
Fax : 408-943-4730
Website : www.cypress.com

© Cypress Semiconductor Corporation, 2007-2014. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.
This Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.
Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.
Use may be limited by and subject to the applicable Cypress software license agreement.