Picoblaze C Compiler

User's Manual 1.1

(2/Jul/2005)

pccomp alpha 1.7.x

Designed by Francesco Poderico
francesco_poderico@yahoo.com

Introduction

Welcome to ***PCCOMP*** (the Picolaze's C Compiler).

If you are a FPGA designer and you are using picoblaze you will be happy to see a C compiler for Picoblaze.

The advantages of using a C compiler rather than assembler are obvious.

1. C is easy.
2. C is a portable language.
3. To debug and write a program in assembler may be very complicated.

The main disadvantage is that a program in C is not “short” as it would be if it was written in assembler

This compiler doesn't do any kind of code optimization. That is because at the moment I want to design something that works correctly, even if is not efficient in terms of speed and code length. When the compiler is completed maybe I will start to design the optimizer.

I hope you will enjoy using “Picoblaze C Compiler”.

Francesco Poderico.

Reporting a bug

If you are reading this document, you have probably accepted to help me in debugging this compiler. At the moment I'm writing this manual for the compiler ver. alpha 1.x.x. I must say that I'm happy to see how things are going, the compiler seems to generate good code even if is still not efficient.

If you find a bug please send an email to "francesco_poderico@yahoo.com" with the subject: ***PCCOMP BUG REPORT***, also describe the bug, and if possible send me the code that has generated the bug. I'll reply when the bug is fixed.

Installation and user interface of pccomp.exe

PCCOMP works on windows machines only, and has been tested on Windows XP and Windows me.

The installation of PCCOMP is very easy.

1. Create a directory called PCCOMP
2. unzip PCCOMP.zip in PCCOMP

Done!

To compile a source file, open a dos shell and go into the same directory where you have installed pccomp.

Type

pccomp [-c] [-v] [-s] file.c <enter>

where the option **-c** is to see your C code commented in the psm file genereted by pccomp.

If you use KCPSM2 then use the option **-v**.

If you use KCPSM3 and **you want to use scratchpad ram** to do stack operation use the option **-s**

If you use KCPSM do not use any options.

ex. pccomp -c test1.c <enter>

pccomp creates a file called test1.psm and also displays any error(s) on the screen.

You can write the errors on an log file using the riderection operator.
For example if you want to write the errors in a file called error.log simple write:
(for example) pccomp -c -s test1.c >>error.log
If there are no errors open test1.psm and have a look.

With PCCOMP you can compile only **one** file at and it MUST contain the main() function.
(we'll see in future that that is not a limitation because you can include files with the #include directive.

PCCOMP generates assembler for KCPSM only!
If you are using the mediatronix IDE then you should translate from the "standard" assembler to the mediatronix assembler from yourself.

THE PREPROCESSOR

PCCOMP has a simple but solid preprocessor. The way the preprocessor has been developed it is easy to expand and more directives maybe added in the future.

The directive that it supports are:
#asm #endasm #ifdef #ifndef #else #endif

The meaning of these directive is the same as in ansi C.

#asm #endasm

The **#asm** directive must be used any time you need to write some code in assembler .

The assembler code should be included between the directive **#asm** and ***#endasm***

for example if you need to do a jump from one part of your program to any other part you can write

```
#asm
        JUMP_HERE:
#endasm


#asm
        jump JUMP_HERE
#endasm
```

When you write some assembler code, pccomp doesn't do any kind of check beetween **#asm** and **#endasm**.

So, is your responsibility to check and be sure the assembler code is correct.
These directive are very useful when writing an interrupt routine. In fact the interrupt routine must be written in assembler only and you should use registers s0,s1,s2 and s3.

#include

#include must be used to include external header file.

The header file in PCCOMP should be viewed a little bit different from ANSI C. In fact in the header file you can write some code. That should allow us (from my point of view) to write some driver directly in the header file. For example if you need a driver that write to an UART, then you can create a header file called UART.H and in it you can write your driver.
At moment I'm writing (2/jul/2005) I'm writing an UART.H driver to allow to write at an UART attached to picoblaze.
In the PCCOMP distribution you should find some header that are:
spartan3.h, sqrt.h.
Have a look in it.
For example in spartan3.h you have 2 function to read and write a single char from the IO space of picoblaze.

Example of use:

```
        #include "spartan3.h"
```

Remember that PCCOMP doesn't do any kind of code optimization, so try to include ONLY the header file that you really need or else you may run out of memory.

#*define*

#define works same as in ANSI C.

you can use this directive when you need conditional compilation or just to keep your code tidy.
One of the non ANSI features of PCCOMP is that you can't declare constant. So instead of that you can use **#define**

Example of use:

```
#define one
#define CONST 10

#ifdef one
         a= a + CONST ;
#endif
#ifndef one
        a = a - CONST;
#endif
```

I recommend to use **#define** when you have a constant value that is used in more part of your code, so when you need to change it you don't need to change all over the program, but you change the numeric value in one place only.

#ifdef, #ifndef, #else #endif

The conditional compilation is an important feature of C, and it may be used with pcomp using the directive **#ifdef**, **#ifndef**, **#else** and **#endif**

example of use:

```
#include "uart.h"
#define board1
// #define board2

void main(){
        #ifdef board1
                printUART("board 1 ready!");
        #else
                #ifdef board2
                        printUART("board 2 ready!");
                #endif
        #endif
}
```

The C Language implementation

The C language for picoblaze is a subset of the ANSI C of K&R. Picoblaze has only 256/1024 byte to allocate it's program and 256 byte of data ram (picoblaze 3 only)

Probably picoblaze will be used to do simple things like i2c, to communicate with an UART, or simple controller, etc.

The biggest issue with picoblaze is probably the program memory, it may run out easily. Especially if you write all the program in C.
There are some tricks to save memory, one is to use *global variable* rather the local variable.
A good programmer prefer to use local variable as much as he cam, but here we are talking of program of 1kbyte of size.
Also using global variable will speed up the execution.
For example try that:

```
int a , b;

void main(){
a = b+1;
}
```

run pccomp the have a look at the generated assembler.

```
        NAMEREG sf , XL
        NAMEREG se , YL
        NAMEREG sd , ZL
        NAMEREG sc , XH
        NAMEREG sa , ZH
        NAMEREG sb , TMP
        NAMEREG s9 , SH
        NAMEREG s8 , SL
        NAMEREG s7 , KH
        NAMEREG s6 , KL
        NAMEREG s5 , TMP2


        CONSTANT        _a_high   ,        ff
        CONSTANT        _a_low    ,        fe
        CONSTANT        _b_high   ,        fd
        CONSTANT        _b_low    ,        fc
        LOAD     YL , fc
        JUMP _main



_main:
        INPUT    XL , _b_low
        INPUT    XH,_b_high
        SUB      YL , 01
        OUTPUT          XH,(YL)
        SUB      YL , 01
        OUTPUT          XL,(YL)
        LOAD ZL,01
        LOAD ZH,00
        INPUT           XL,(YL)
        ADD      YL , 01
        INPUT           XH,(YL)
        ADD      YL , 01
        ADD      XL , ZL
        ADDCY    XH , ZH
        OUTPUT XL ,_a_low
        OUTPUT XH ,_a_high
        RETURN
```

now try to compile the following code and compare the psm files.

```
void main(){
int a , b;
a = b+1;
}
```

```
            NAMEREG sf , XL
            NAMEREG se , YL
            NAMEREG sd , ZL
            NAMEREG sc , XH
            NAMEREG sa , ZH
            NAMEREG sb , TMP
            NAMEREG s9 , SH
            NAMEREG s8 , SL
            NAMEREG s7 , KH
            NAMEREG s6 , KL
            NAMEREG s5 , TMP2



_main:
            SUB     YL , 04
            LOAD    XL,         YL
            ADD     XL ,        02
            SUB     YL , 01
            OUTPUT              XL,(YL)
            LOAD    XL,         YL
            ADD     XL ,        01
            INPUT               ZL , (XL)
            ADD     XL , 01
            INPUT               ZH, (XL)
            ADD     XL , 01
            SUB     YL , 01
            OUTPUT              ZH,(YL)
            SUB     YL , 01
            OUTPUT              ZL,(YL)
            LOAD ZL,01
            LOAD ZH,00
            INPUT               XL,(YL)
            ADD     YL , 01
            INPUT               XH,(YL)
            ADD     YL , 01
            ADD     XL , ZL
            ADDCY   XH , ZH
            LOAD    ZL , XL
            INPUT               XL,(YL)
            ADD     YL , 01
            OUTPUT              ZL,(XL)
            ADD     XL , 01
            OUTPUT              ZH,(XL)
            ADD     YL , 04
            LOAD    XL , ZL
            LOAD    XH , ZH
            RETURN

            LOAD    YL , 00
```

See? This file is much larger that the first one.
The reason of that is because when pccomp creates a local variable needs to allocate some space on the data stack. In this case it needs to allocate 4 bytes to keep the variables a and b.
The access to a and b is slow because any time pccomp needs to see where is a or b he need to calculate.
When a variable is declared globally, the access is immediate because the compiler knows excactly where is located and it doesn't need to recalculate any time.

When you write
a = b + 0;

pccomp is not clever enough to understand that what we want is “ a = b;”
and generate the code to do an addition between b and 0 and save the result in a!
You may have this problem when you write code like:

#define my_const 0

```
char a;
void main(){
a =  a + my_const;}
```

Another example is a = a+2-(3*4) + 1;
I strongly do not recommend to write code like that!
pcomp will put 1 on the stack after 4 and 3 will calculate 3*4 and the result will go on the stack and added to 1 and put on the stack again and added to 2 and putted on the stack again and then will put a on the stack, and calculate a+2 and saves the result in a!!!!
This is really not efficient.
So we learned that pccomp has not build inside any kind of optimizer.

In theory a compiler can be divided in two category stack and RISC optimized.Design a stack like compiler is much easier than a RISC optimized.Because this kind of compiler don't care about the CPU architecture. Even the debug is easier.PCCOMP is one of this... is a STACK compiler.. easy but working! (that is what I want)

Read carefully this user manual and you should be able to use this compiler without problems. If you find a bug, please send me an email.

I usually reply to everybody in less then 5 days unless I can't.

Type

PCCOMP supports at moment the following types:

char (-127 +128) 8 bit

int (-32727 +32768) 16 bit

unsigned int (0 .. +65535) 16 bit

unsigned char (0 .. 255) 8 bit

*Some of you may complain that the **long** and the **float** types are not implemented, I may implemented them one day maybe. On the other side if you need to use a float, maybe your program is quite complicated and picoblaze may not suite for that, for example imagine how long it may take a multiplication between 2 float in software.*

You can declare a variable as global and as local.

Global variable should be declared at the **top of your program** and not anywhere, this is also a good practice.
The same rule is true for the local variables them should be declared **at the top of the function**.
We know that in C you can declare a variable (in practice) where you want. This is NOT the case of PCCOMP.
from this point of view it looks a bit more like pascal

*REMEMBER:With PCCOMP the variable **must** be declared only on the t**op of your function** and/or on the t**op of your code.***

I just said that PCCOMP has not an in build optimizer, so I'm giving you now some tips to “save space” .
Every time you declare a variable a compiler allocate same space in the memory.
When we declare a global variable the variable is there all the time, and the compiler knows exactly where is located.
A different matter is for the local variables. A local variable exist only for the time that the function (where is declared) exists (remember that even “main()” is a function).

So the compiler doesn't know were is located at the time of compilation, because could be anywhere.
For that reason a local variable must be calculate at the execution time. That cause the creation of code to manage that.

if you want to write some code that is efficient I strongly suggest you to declare the variable globally.

Example of type declarations:

***int** i;* *// integer 16 bits*

***char** a,ii,aa[10];* *// char +char + array of 10 elemnts of a char (12 bytes)*

***unsigned char** q;* *// unsigned char 8 bits*

***unsigned int** w;* *// unsigned integer 16bits*

***int** *p;* *// pointer to integer 8 bit*

***int** pp[10];* *// array of 10 integer (20 bytes)*

Example of use

***for**(ii =0;ii<10;ii++) a[ii] = a[ii] +1; // OK!*

***for**(i =0;i<10;i++) a[i] = a[i] +1; // WRONG*

Q: Why the first for is correct and the second is wrong?

If you see above ii is declared a char, but i is declared as integer. Because Picoblaze 3 has a physical limit of 256 (the scrathcpad ram) and we don't want to use more memory that we need do we?Also if you think you may need an array longer that 256 bytes.... maybe you need microblaze or the PPC.In that case Xilinx has designed a cool tool called EDK (that I'm currently using at work) that is very cool nad

p = 0x00; *// pointer at address 0x00*

*p = 1; *// write 1 at address 0x00*

p = &pp; *// p point at the first address of the array pp*

p = &pp[0]; */WRONG PCCOMP doesn't allow that*

Pointer and array

How we saw on the examples above we can declare pointer and array, with some limitations.

We can use only array with 1 dimension only.

The maximum array length is 256, and the index must be a char or an unsigned char.

This is not a real limitation because you can addresses up to 256 elements. Also the fact that you must use a char as index for the array allow you to save some memory.

Pointers are very useful in C, on the next pages you can read how to use the pointers to do IO with the FPGA. Also the pointer has some limitations;

p = &pp; // OK

p = &pp[0]; // WRONG is ANSI C is OK but with PCCOMP is illegal.

With C you can virtually do anything, but you with PCCOMP you have some limitation.

The pointer can point only to variables (global or local) and to array.
You can't have pointer to pointer.
For example:

```
 char ** p; // This is illegal
```

When you point at an array be sure to use a syntax like: ***pointer = &array***, (see example above) else the compiler gives not working code.
I never wrote code like: p=&pp[i]; I think is horrible! So I don't think that is a big issue for everybody.
Bu just keep in mind that is not allowed to point at generic element of an array.

Casting between different types

When you have two or more variable of different type that must be added or divided (for example) between them you should use the casting operator first. The casting is very useful and you can cast from:

1. **char** -> **int**
2. **char** -> **unsigned int**
3. **unsigned char** -> **int**
4. **unsigned char** -> **unsigned int**

The points 1, 3 and 4 doesn't need any precaution.
The point 2 it need some precaution, because the signed number (char can be consideted us a signed integer on 8 buts) of -1 is “11111111” that is 255.

example :

```
char a;
unsigned int b;

void main(){
b = 0;
a = -1;
b = (unsigned int) a;

}
```

When we compile this code with PCCOMP and see the psm file we can notice that b is 255 now!
This is not a fault of the compiler, this is just bad coding. So, when you need casting, just be careful.

```
        CONSTANT        _a              ,       ff
        CONSTANT        _b_high   ,     fe
        CONSTANT        _b_low    ,     fd
        LOAD    YL , fd
        JUMP _main
;char a;
;unsigned int b;
;void main(){
_main:
;b = 0;
        LOAD ZL,00
        LOAD ZH,00
        OUTPUT  ZL,_b_low
        OUTPUT  ZH,_b_high
;a= -1;
        LOAD ZL,ff
        OUTPUT  ZL,_a
;b = (unsigned int) a;
        INPUT   ZL ,_a
        OUTPUT  ZL,_b_low
        OUTPUT  ZH,_b_high
;}
        LOAD    XL , ZL
        LOAD    XH , ZH
        RETURN
```

You may fall in a similar problem in expressions like:

```
unsigned char a;
char b;
```

```
void main(){
b = -1;
a =1;
if a>b // .... 1 > -1 yes.. but the left opetar is an unsigned char and the right is a signed ???? is bad coding!
 a = 0xaa;
else
a = 0xbb;
}
```

PCCOMP allows casting, but has not all the features of ANSI C.
When you write code like

i = 5;
the number 5 must be read from the compiler as integer if i is an integer, as a char if i is a char and so on. The way PCCOMP works it use the last value putted on the stack to understand the type of the number. That imply some attention.

Example this code is OK

```
        char i;

        int ss;

        void main(){

                ss = (int)(i - 5); // Ok!

        }
```

i is a char and 5 is interpeted as char. Then the difference i-5 (still a char) is casted in integer thats fine, but if you write instead:

```
        char i;
        int ss;

        void main(){

                ss = (int)i - 5; // WRONG

        }
```

This code is wrong because i is a char, so the compiler bealive that we need a char, but in reality we have converted i from char to integer. So pccomp generates code to do a subtraction between 2 char. Unfortunatley PCCOMP doesn't gives any warning of that. So you need to be carefully.
When you are not sure try to be “safe” as possible. For example you can write:

```
        char i;
        int ss;

        void main(){

                ss =(int) i;

                ss -=5; // OK;

        }
```

IF ***statement***

The syntax of the if statement is: **if**(<bool_expr>)<expr>[**else**]<expr>

ex.

```
if( i<=10) {
        read = inchar(ff);
        i++;
}

if (i <99)
        ii = ii +1;
else
        ii = ii -1;
```

WHILE ***statement***

The syntax is: **while**(<bool_exp>) <expr>

ex.

a good example is the infinite loop.

```
while(1) {do dome ops} // this is wrong because 1 is not a boolean expressions
```

but ifyou write

```
while(1>0) {do some ops} // thi is OK because 1>0 is a boolean expression
```

If you need to do an infinite loop the most efficient way is NOT to use the while expression, but write your code half in asm and half in C.

ex.

```
#asm
        while1loop:
#endasm
        // your code here
#asm
        jump while1loop
#endasm
```

DO statement

The syntax of the do statement is: do<expr> while(bool_exp>)

example

```
do{
        for(i=1;i<10;i++) c[i] = a[i]*b[j];
        j++;
    }while(j<10)
```

FOR statement

The syntax is: **for**(<expr>;<expr>;<expr>) <expr>

example;

```
    for(i=1;i<10;i++) c[i] = a[i]*b[j];
```

SWITCH statement

The syntax is **switch**(<expr>) { **case** <const> : <expr>;[**break**] [case <const> : <expr>;[break]]}

ex:

```
    swtich(channel) {
        case 'a' :      rr++;
            break;
        case 'b':       rr--;
            break;
        default :
            rr =0;
    }
```

INTERRUPTS

Like any other compiler pccomp supports interrupts. Picoblaze has only one external interrupt line that can be enabled or disabled.

The command to enable the interrupt is ***IRQ_ON; // do not forgot at the end of line ";"***

The command to disable the interrupt is ***IRQ_OFF;***

The interrupt routine must be written in assembler, and you should use only the registers s0,s1,s2,s3.

The interrupt must be declared and defined, as any other functions.

The declaration of an interrupt routine is **interrupt** my_irq(**void**);

Example: we want to do the sum of a + b when when have an external interrupt request;

```
        interrupt my_irq(void);
        char a,b,c;
        char had_an_irq;
        void main(){
                IRQ_ON;
                while (1>0)
                        while(had_an_irq) {
                                IRQ_OFF;
                                had_an_irq = 0;
                                c=a+b;
                                IRQ_ON;
                        }
        }
        interrupt my_code(void){
                #asm
                        load s0,1
                        store s0,_had_an_irq ; had_an_irq = 1
                #endasm
        }
```

To Remember

Picoblaze has two way to end and interrupt routine:

RETURNI ENABLE

RETURNI DISABLE

For default the version alpha 1.x.x has RETURNI DISABLE.

.

FUNCTIONS

the parameter of function can not be pointer or array.

Any function must be declared and defined.

Example:

```
        char sum (char a, char b);
        char a,b,apb;


char sum(char a, char b){
                return  a +b;
                }

void main()
                {
                .
                .
                apb = sum(a,b);
                }
```

Examples

I've included with the compiler some tests examples.
I was asked to generates more examples, and I'm doing so.
I will appreciate if you could send me any C code example that could be interesting to add in this sections. For example if you have done some I2C controller or an LCD driver, etc.

SPARTAN3.h

In spartan3.h you will find 2 routines useful when you need to do IO and you are using picoblaze 3.

The library has 2 functions:
1 **char** inchar(**char** addr)
2 **void** outchar(**char** addr, **char** data)

The function inchar is used to read a char from the IO space at address addr.
For example:

char xx;

// some code

xx = inchar(0x01);// here we are reading at the IO space 1

when we need to write a char in a specific location we may write code like:

outchar(0x01, xx);
In this case we wrote xx at the IO 0x01.
It may be interesting to have a look at the spartan3.h library, and I copied here for your interest.

```
void outchar(char addr, char data); // write a character to the io space
char inchar( char addr);                    // read a character from the io space
//
//
//
//
/////////////////////////////////////
// READ FUNCTION
char inchar(char addr){
          char x;
        #asm
                LOAD TMP , YL
                ADD  TMP , 01
                FETCH TMP, (TMP) ; TMP = addr
                INPUT TMP,(TMP) ; TMP = datain
                STORE TMP,(YL) ; save in x

        #endasm

        return x;
        }
//
//
//
//
//////////////////////////////////////
// WRITE FUNCTION

void outchar(char addr , char data){
        #asm
                FETCH TMP,(YL) ; TMP = data
                LOAD TMP2,YL
                ADD TMP2 , 01
                FETCH TMP2,(TMP2) ; TMP2 = addr
                OUTPUT TMP,(TMP2)
                #endasm
//
//
// END LIBRARY SPARTAN3.H
//////////////////////////////////////



}
```