

## SUMMARY

- Why C compiler?
- HW and SW tools: two separate projects?
- Compiler overview
- Conclusions



# Why C compiler?



## Time to market rules

### Writing applications using assembly language:

- Gives you great performances (direct access to the HW)

But...

- It takes a LOT of time
- It requires very skilled, specialized developers
- Difficult to maintain, reuse, etc...

### Writing applications using C language:

- Faster development time
- Easier to train (and reuse) developers
- Easier to maintain, reuse, etc...

But...

- The compiler **MUST** provide great performances

## Time to market rules (2)

- Developing applications in C is faster:  
Real life example: DSP library development effort (skilled team, similar functions efficiency):
  - mAgic (optimizing assembler):  
70 functions: 31 person/months
  - mAgicV (C compiler):  
80 functions: 12 person/months

Gain factor using C: ~3x



## Time to market rules (3)

- Writing C enables the use of less specialized, easier to form and more reusable teams:

C language	Assembly language
<ul style="list-style-type: none"><li>• C common constructs<ul style="list-style-type: none"><li>• For.. Next</li><li>• If.. Then.. Else</li></ul></li><li>• C common data types</li><li>• custom data types</li><li>• Intrinsic</li><li>• Light architecture knowledge:<ul style="list-style-type: none"><li>• Parallelism</li><li>• Memories</li></ul></li></ul>	<ul style="list-style-type: none"><li>• Lots of specific assembly instructions<ul style="list-style-type: none"><li>• Arithmetics</li><li>• Memory moves</li><li>• Registers moves</li><li>• Strange ones...</li></ul></li><li>• Specific register types and relationships</li><li>• Deep architecture knowledge:<ul style="list-style-type: none"><li>• Parallelism</li><li>• Registers</li><li>• Memories</li><li>• Pipelines</li><li>• ...</li></ul></li></ul>

Reusable skills in blue, not reusable ones in red

- Allows easier reuse of legacy code
- Easier to maintain

# HW and SW tools: two different projects?

## HW and SW tools: a global project

“Make the HW, SW will follow!”

Quite an unrealistic sentence nowadays:

- The SW tools development could cost more than HW development
- Small modifications in HW could have a great impact on SW tools development (“don’t do in HW what could be done in SW” it’s NOT always a law!)
- You could have the most powerful HW, but if your SW tools can’t use it, you could have better spent your money
- Your customers expect from you the best HW **AND** the best SW tools

But...

- HW must be as simple as possible (lower cost, higher freq, less area,..)
- HW must be competitive on the market

## HW and SW tools: a global project

“Think to (and possibly develop) the SW tools while you develop the HW”

You will gain a lot:

- SW tools development will cost less
- All the wonderful capabilities of your HW will be available also to your customers
- At the end you will have the best HW and the best SW tools you can afford with your budget
- You can easily understand if one HW feature is really important for your applications (if not, throw it away)
- For sure, with good tools, your product will be more competitive on the market

But...

- SW tools must not be a limitation for developing an HW that must be competitive!



## HW and SW tools: a global project

Real life: 2 banks memory in mAgic and mAgicV

- mAgic and mAgicV DSPs internal memory is divided into two banks (vector and complex data access)
- The core can see memory addresses in two ways:

Bank 0	Bank 1
Address 0	Address 0
Address 1...	Address 1...

Or

Bank 0	Bank 1
Address 0	Address 1
Address 2...	Address 3...

- First approach: 2 different ops for bank0 and bank1
- Second approach: only 1 op for both banks: bank is selected in HW by address parity

Same performances, same bandwidth but...

- Second approach greatly simplify SW tools development
- Maybe with the first approach your tools could not even be able to fully exploit the capabilities of your HW

**Result: little HW effort, great SW tools simplification**



## HW and SW tools: a global project

### Real life: memory addresses generation in mAgicV

- Due to their complexity, VLIW DSPs could have a lot of different addressing modes
- Our target for mAgicV was:
  - Simplify as much as possible the HW of the AGU
  - Maintain performances
  - Maintain programmability
- Strategy:
  - Define the AGU I/S with C programmability in mind
  - Try it on important DSP kernels with the C compiler
  - Cut away all the modes not useful or impossible to implement in C

**Result: no performance loss with respect to HW architecture, easier to implement/debug HW, 100% performance directly in C**

## HW and SW tools: a global project

Real life: Software Pipelining HW support, predication

- mAgicV offers in HW a support for saving program memory when doing SW Pipelining, and the support for predicated execution
- The C compiler doesn't support them up to now (it will in a near future). They are available in assembly
- **Question:** how many customers will decide to switch to assembly coding because of these features?
- **Answer:** probably 0, but they will complain to you because they would like to have them supported in C

**Result: C programming gives a terrible advantage. If the C compiler gives you the needed performances and a particular, not crucial HW feature can't be exploited in C, it will probably remain unused.**

**Two solutions: put it in the compiler or throw it away**

## Compiler overview

# Compiler overview

## Considerations on nML language

- The mAgicV compiler and tools were developed using Target Compiler Technologies nML based Chess/Checkers toolchain
- Modeling with nML enable the user to describe a very wide range of architectures: all of the mAgicV features can be described with it.
- All the features described are effectively used by the compiler.
- Very easy methodology for describing custom data types (like complex or vector) using classes and compiler known functions.
- Overall, the effort required for obtaining a high performance prototype compiler is extremely reduced
- Performance obtained are comparable with hand-optimized assembler
- The same model also produces assembler, linker and cycle-accurate simulator, providing a fully integrated programming environment.

## Considerations on Chess compiler

- The compiler allows a very smart usage of the assembly instructions, also of the strangest ones. This is clearly demonstrated by the mAgicV model.
- Support for most of the classical processor independent optimizations.
- Very efficient function calling, with register based parameter passing.
- Back end optimizations includes a powerful SW pipelining engine.
- Due to the compiler structure, compile times are long (order of minutes)

## mAgicV C compiler (1)

- **Compiler capability of handling 'vanilla' code**
  - Most code performs near max using `restrict` and just two loop annotations: `chess_loop_range(min,max)` to annotate the minimum number of loop iterations and `chess_prepare_for_pipelining` for allowing the generation of sw pipelined loops that needs also an epilogue.
  - Support for register file bypass.
- **Support for all the C and mAgicV native data types and memory hierarchy**
  - Supported types:
    - **char:** 16bit registers, 40/32bit memory (int/ext)
    - **int:** 16bit registers, 40/32bit memory (int/ext)
    - **long,long long:** 32/40bit registers, 40/32bit memory (int/ext)
    - **float,double:** 40bit registers, 40/32bit memory (int/ext)
    - **\_c\_float,\_v\_float:** 2\*40bit registers, 80/64bit memory (int/ext)
    - **\_c\_long,\_v\_long:** 2\*40bit registers, 80/64bit memory (int/ext)
  - Supported memories:
    - **Internal data memory:** 16kw \* 40bit
    - **System peripherals:** ARM amba bus mapped peripherals and external memory

## mAgicV C compiler (2)

- **Support for circular addressing**

- Circular pointers are supported for internal and external memories using `chess_storage` with reserved registers. This allows to initialize the length, stride and base address using normal int variables:

```
//circ pointer initialization
static int chess_storage(A14) MyPtr = &X;           //circular pointer
static int chess_storage(M14) MyPtrMod = 10;        //pointer modifier
static int chess_storage(L14) MyPtrLen = 100;       //pointer length
static int chess_storage(S14) MyPtrBase = (int)X;   //pointer base
```

- **Support for interrupt handling**

- Interrupt service routines can be declared using the `property(isr)` attribute:

```
//interrupt service routine
void vIsrRoutine() property(isr){
    ...
}
```

- Interrupt service routine can use also `property(envelope)`. This automatically saves and restore the whole context before and after the function, good for big and complex interrupt handlers

## mAgicV C compiler (3)

- **Support for external memory/ARM amba bus memory mapped peripherals**
  - Data from and to external resources must first be moved in internal memory using DMA, than can be moved to register files. The access to all the memory mapped resources, including external memory, is available using variables declared with the attribute `chess_storage(EXT_DATA[:addr])`

Access to peripherals:

```
//declare a struct type for interacting with a USART peripheral
struct AT91S_USART {AT91_REG US_CR; ...};
//declare a struct AT91S_USART pointer to the AMBA memory mapped USART 0
AT91S_USART chess_storage(EXT_DATA) *pUsart0 =
    (AT91S_USART chess_storage(EXT_DATA)*)AT91C_BASEADDR_US0;
AT91S_USART MyUsart; //declare a struct AT91S_USART in internal memory
MyUsart=*pUsart0; //copy the memory mapped struct to internal memory
MyUsart.US_CR=1; //modify the USART configuration
*pUsart0=MyUsart; //write it back into the peripheral bus mapped registers
```

Access to external memory:

```
//single location or vectors handling using intrinsics
float chess_storage(EXT_DATA:XM_BASE+1000) a[100];
float b[100];
_READEXT(a,b,100); //reads 100 values from external to internal memory
b[19]=0.0; //modify the vector
_WRITEEXT(a,b,20); //write back to external memory only first 20 values
```

## mAgicV C compiler (4)

- **Supported intrinsics**

- The compiler supports 59 intrinsic functions that allows to use any basic arithmetic operation on long, float, vector and complex data types. Using them, it is possible to use C plus intrinsics as a kind of macro assembler. These are also intrinsics dealing with component access of vector and complex types, and external memory access. A few examples:

```
//complex multiplication with conjugate second operand
_c_float in1,in2;
_c_float res=_CJMUL(in1,in2);
```

```
//selection based on condition
float selected,a,b;
static bool chess_storage(FLACKCOND0) cond=(a<10);
selected =_SEL(cond,a,b);
```

```
//modify real and imaginary part of a complex number
_c_long num;
long new_real=10, new_imag=20;
_SETREAL(num,new_real);
_SETIMAG(num,new_imag);
```

- **Support for code compression**

- The mAgic code compressor is integrated within the mAgicV compile chain

## Status of mAgicV C compiler (5)

- **Available libraries:**

- D940 DSP lib: 110 DSP kernel functions all developed in C
- Math library: all developed in C
- Libc: porting of DietLibc, a light C runtime for embedded (except memory management routines that were developed internally for efficiency)
- DBIOS lib: a collection of functions that can be used to access system peripherals and facilitate ARM – mAgicV interaction

## Status of mAgicV C compiler (6)

- **Available documentation:**
- Compiler user manual (including IDE)
- Linker user manual
- Assembler user manual
- Cycle accurate simulator user manual
- mAgic specific C compiler (data types, memories) PRELIMINARY
- mAgic parallel assembly syntax PRELIMINARY



## Target compiler status

# Examples of typical DSP kernels performances





## DSP kernels comparison

function	Theoretical Performance on D940	D940 C Compr factor ~1.5	TI C67 family hand optimized assembly (from web site)
Vector Sum	Unroll 4: $0.875 \times N + K$	Unroll 4: $0.875 \times N + 11$ 192 bytes	$1 \times N + 8$ 192 bytes (based on vecmul size)
Vector Complex Product	Unroll 4: $1.5 \times N + K$	Unroll 4: $1.5 \times N + 12$ 192 bytes	$2 \times N + 18$ (estimated)
Complex FIR	Unroll 4: $(K' + M) \times L + K$	Unroll 4: $(24+M) \times L + 13$ 448 bytes	$(2 \times M + 14) \times L + 17 + (L-1)$ 640 bytes
Vector Division	$3 \times N + K$	$3 \times N + 24$ 320 bytes (40 bit precision) Computing $y/x$	$8 \times ((N-1)/4) + 53$ 512 bytes (32 bit precision) Computing $1/x$
Vector SQRT	$7 \times N + K$	$7 \times N + 39$ 576 bytes	



## Code example: vector sum

```

void vadd_vector_unroll4(_v_float * restrict xp, _v_float * restrict yp,
                        v_float * restrict zp, unsigned int leng){
#ifdef _CIRC_POINTERS_
    static unsigned int chess_storage(S0) s0;
    static unsigned int chess_storage(S1) s1;
    static unsigned int chess_storage(L0) l0;
    static unsigned int chess_storage(L1) l1;
    s0=(unsigned int)xp;s1=(unsigned int)yp;l0=leng;l1=leng;
#endif
    unsigned int i;
    for(i=0; i<leng; i++) chess_unroll_loop(4) chess_loop_range(1, ) {
        *zp++=*xp++ + *yp++;
    }
#ifdef _CIRC_POINTERS_
    s0=0;s1=0;l0=0;l1=0;
#endif
}

```

```

0  1.M=2 : - - : TMP1=2 -
2  loop_counter=3.A : - - : - -
1  0.M=1.M : - - : 3.A=3.A A>>TMP1 -
3  REPEAT 6 : - - : - -
4  - : VRF20 = VDATA[1.A;1.A+=1.M] : - : VRF16 = VDATA[0.A;0.A+=0.M] : -
5  - : VRF12 = VDATA[1.A;1.A+=1.M] : - : VRF10 = VDATA[0.A;0.A+=0.M] : -
6  - : VRF6 = VDATA[0.A;0.A+=0.M] : - : VRF8 = VDATA[1.A;1.A+=1.M] : -
7  - : VRF14 = VDATA[0.A;0.A+=0.M] : - : VDATA[2.A;2.A+=0.M] = VRF22 = CFADD(VRF16,VRF20)
8  - : VRF16 = VDATA[0.A;0.A+=0.M] : - : VDATA[2.A;2.A+=0.M] = VRF24 = CFADD(VRF12,VRF10)
9  - : VRF18 = VDATA[1.A;1.A+=1.M] : - : VDATA[2.A;2.A+=0.M] = VRF26 = CFADD(VRF6,VRF8)
10 - : VRF10 = VDATA[0.A;0.A+=0.M] : - : VRF20 = VDATA[1.A;1.A+=1.M] : -
11 - : VRF12 = VDATA[1.A;1.A+=1.M] : - : - -
12 - : VRF6 = VDATA[0.A;0.A+=0.M] : - : VDATA[2.A;2.A+=0.M] = VRF28 = CFADD(VRF18,VRF14)
13 - : VRF8 = VDATA[1.A;1.A+=1.M] : - : - -
14 RETURN : - - : - -
15 - : - - : - -
16 - : - - : - -
17 - : - - : - -

```

## Code example: Complex Fir

```

void fir_unroll4(_c_float * restrict x, _c_float * restrict h,
                _c_float * restrict y, unsigned int xLEN, unsigned int hLEN){
    unsigned int n, k;
    _c_float acc1,acc2,acc3,acc4;
    for(n=hLEN-1; n<xLEN; n++) chess_loop_range(1, )
    {
        temp = _c_float(0.0,0.0);
        temp1 = _c_float(0.0,0.0);
        temp3 = _c_float(0.0,0.0);
        temp4 = _c_float(0.0,0.0);
        _c_float * restrict pp=x+n;
        _c_float * restrict ppl=h;
        for(k=0; k<hLEN/4; k++) chess_loop_range(1, )
        {
            temp = temp + (*pp--) * (*ppl++);
            temp1 = temp1 + (*pp--) * (*ppl++);
            temp2 = temp2 + (*pp--) * (*ppl++);
            temp3 = temp3 + (*pp--) * (*ppl++);
        }
        y[n-hLEN+1] = (temp+temp1)+(temp2+temp3);
    }
}

```

```

...
21 REPEAT 3 : - - : - -
22 - : VRF30 = VDATA[1.A;1.A-=1.M] : VRF14 = VRF6 : VRF34 = VDATA[6.A;6.A+=6.M] : -
23 - : VRF26 = VDATA[6.A;6.A+=6.M] : VRF38 = CFMUL(VRF32,VRF36) : VRF20 = VDATA[1.A;1.A-=1.M] : VRF8 = CFADD(VRF6,VRF0)
24 - : VRF16 = VDATA[1.A;1.A-=1.M] : VRF12 = VRF6 : VRF22 = VDATA[6.A;6.A+=6.M] : VRF10 = CFADD(VRF6,VRF0)
25 - : VRF36 = VDATA[6.A;6.A+=6.M] : VRF18 = CFMUL(VRF34,VRF30) : VRF32 = VDATA[1.A;1.A-=1.M] : VRF14 = CFADD(VRF18,VRF14)
26 - : VRF34 = VDATA[6.A;6.A+=6.M] : VRF24 = CFMUL(VRF26,VRF20) : VRF30 = VDATA[1.A;1.A-=1.M] : VRF8 = CFADD(VRF24,VRF8)
27 - : VRF26 = VDATA[6.A;6.A+=6.M] : VRF28 = CFMUL(VRF22,VRF16) : VRF20 = VDATA[1.A;1.A-=1.M] : VRF12 = CFADD(VRF28,VRF12)
28 - : VRF22 = VDATA[6.A;6.A+=6.M] : VRF38 = CFMUL(VRF36,VRF32) : VRF16 = VDATA[1.A;1.A-=1.M] : VRF10 = CFADD(VRF38,VRF10)
...

```

## Code example: Vector division

```

void vdivv(_v_float * restrict a, _v_float * restrict b,
          _v_float * restrict result, unsigned int len){
    _v_float two=_v_float(2.0,2.0);
    _v_float tmp1,tmp2,tmp3,tmp4,tmp5,tmp6,tmp7,va11,va12;
    va11=*b++;
    va12=*a++;
    tmp1=_INVSEED(va11);
    for (unsigned int i=0;i<len;i++) chess_loop_range(1,) {
        tmp3=tmp1*va12;
        tmp2=tmp1*va11;
        tmp4=two-tmp2;
        tmp5=tmp4*tmp2;
        tmp6=tmp4*tmp3;
        tmp7=two-tmp5;
        *result++=tmp7*tmp6;
        va11=*b++;
        va12=*a++;
        tmp1=_INVSEED(va11);
    }
    #if 0
    //note that this simple code is enough to give 80% performances
    for (unsigned int i=0;i<len;i++) chess_loop_range(1,) {
        *result++ = (*a++) / (*b++);
    }
    #endif
}

```

```

...
11 REPEAT 5 : - - : - -
12 - : - VRF0xc = VINVSEED(VRF0xa) : - -
13 - : - VRF0xe = VFMUL(VRF0x8,VRF0xc) : - VRF0x10 = CFADD(VRF0x6,-VRF0x14)
14 - : VRF0x8 = VDATA[0x0.A;0x0.A+=0x0.M] : VRF0x12 = VFMUL(VRF0xe,VRF0x10) : - VRF0x18 = CFADD(VRF0x6,-VRF0x16)
15 - : VRF0x8 = VDATA[0x0.A;0x0.A+=0x0.M] : VRF0x14 = VFMUL(VRF0xa,VRF0xc) : VRF0xa = VDATA[0x1.A;0x1.A+=0x1.M] : -
16 - : - VRF0x16 = VFMUL(VRF0x10,VRF0x14) : - -
17 - : - VRF0xe = VFMUL(VRF0xc,VRF0x8) : - -
18 - : - VRF0xc = VINVSEED(VRF0xa) : - VRF0x10 = CFADD(VRF0x6,-VRF0x14)
19 - : - VRF0x12 = VFMUL(VRF0x10,VRF0xe) : - VRF0x18 = CFADD(VRF0x6,-VRF0x16)
20 - : VDATA[0x2.A;0x2.A+=0x2.M] = VRF0x1a = VFMUL(VRF0x18,VRF0x12) : - -
...

```



## Target compiler status

# Conclusions

## Conclusions

- **Atmel Roma paradigm: mAgicV DSP is programmable with excellent performances using only C language: this allows a great reduction of time to market in innovative application development and facilitates porting of existing codes.**
- The majority of the implemented DSP kernels deliver very high performances
- DSP and Math library have been developed in C, no need for assembly
- Easy access to peripherals and external memory allows writing BIOS functions directly in C
- 'Vanilla' code could be quite easily ported to mAgicV (difficulties: int is 16bit, unsigned long is emulated, no data cache)

# Thank you!