ESTIMATION OF WCET USING A LITTLE LANGUAGE TO DESCRIBE MICROCONTTROLER AND DSP ARCHITECTURES

Adriano José Tavares
Department of Industrial Electronics, University of Minho, 4800 Guimarães, Portugal
e-mail: atavares@dei.uminho.pt, phone: (351) 253 604706

Carlos Alberto Couto
Department of Industrial Electronics, University of Minho, 4800 Guimarães, Portugal
e-mail: ccouto@dei.uminho.pt, phone: (351) 253 604701

ABSTRACT

A method for analysing and predicting the timing properties of a program fragment will be described. First a little language implemented to describe a processor's architecture is presented followed by the presentation of a new static WCET estimation method. The timing analysis starts by compiling a processor's architecture program followed by the disassembling of the program fragment. After sectioning the assembler program into basic blocks call graphs are generated and these data are later used to evaluate the pipeline hazards and cache miss that penalize the real-time performance. Some experimental results of using the developed tool to predict the WCET of code segments using some Intel microcontroller are presented. Finally, some conclusions and future work are presented.

1. INTRODUCTION

Real-time systems are characterized by the need to satisfy a huge timing and logical constraints that regulate their correctness. Therefore, predicting a tight worst case execution time of a code segment will be a must to guarantee the system correctness and performance.

The simplest approach to estimate the execution time of a program fragment is:

1. for each arithmetic instruction, counting the number of times it appears on the code
2. express the contribution of this instruction in terms of clock cycles,
3. update the total clock cycles with this contribution.

Other two basic approaches are:

1) Isolate the operation to be measured and make time measurements before and after performing it, which is valid only when the resolution of an individual measurement will be considerably less than the time of the operation to be specifically analysed
2) Execution of the operation a large number of time, and at the end of the loop operation execution, the desired time will be found by averaging. Even with this approach, if you want an accurate measurement, a number of complications such as, compiler optimisations, operating system distortions, must be solved.

Nevertheless, these approaches are unrealistic since they ignore the system interferences and the effects of cache and pipeline, two very important features of some processors that can be used in our hardware architecture. Shaw [1], Puschner [2], and Mok [3], developed some very elaborated methodology for WCET estimation, but none of them takes into account the effects of cache and pipeline.

Theoretically, the estimation of WCET must skip over all the profits provided by modern processors, such as caches, and pipeline (i.e., each instruction execution suffers from all kind of pipeline hazards and each memory access would cause a cache miss) as they are the main source of uncertainty. Experimentally, a very pessimistic result would be obtained, and so, making useless those processor's resources. Some WCET estimation schemes oriented to modern hardware features, were presented in the last years, and among them we refer to: Bharrat [4], Nilsen [5], Steven Li [6], Zhang [7], Tai-Yi Huang [8], Whalley [9], and Sung-Soo Lim [10]. However, these WCET estimators do not address some specificity of our target processors (microcontrollers and DSPs), since they are oriented to general-purpose processor. Therefore, we propose a new machine independent estimator, implemented as a little language for architecture description. Such a machine independent scheme, based on the little language was used before by Scharr [11] to describe the pipeline instruction scheduling and executable editing, Tremblay[12] to generate machine independent code, Proebsting and Fraser [13] to describe pipeline architectures and Nilsen [5] to implement a compiler, simulator and WCET estimator for pipeline processors.

2. LITTLE LANGUAGE PROCESSOR

Fig.1 shows the language processor organization of the implemented little language. The purpose of a little language, typically, is to solve a specific problem and, in so doing, simplify the activities related to the solution of the problem. Our little language's statements are created based on the tasks that must be performed to describe processor's architectures in terms of structure and functional architecture of the interrupt controller, PTS (Peripheral Transaction Server), PWM (Pulse Width Modulation), WG (Waveform Generator), and HIS (High Speed Input), instruction set, instruction semantics, addressing modes, processor's registers, instruction coding, compiler's specificity, pipeline and cache resources, and so on. Strongly related to the instruction's semantics of a little language is the language paradigm that defines how the language processor must process the built-in statements.

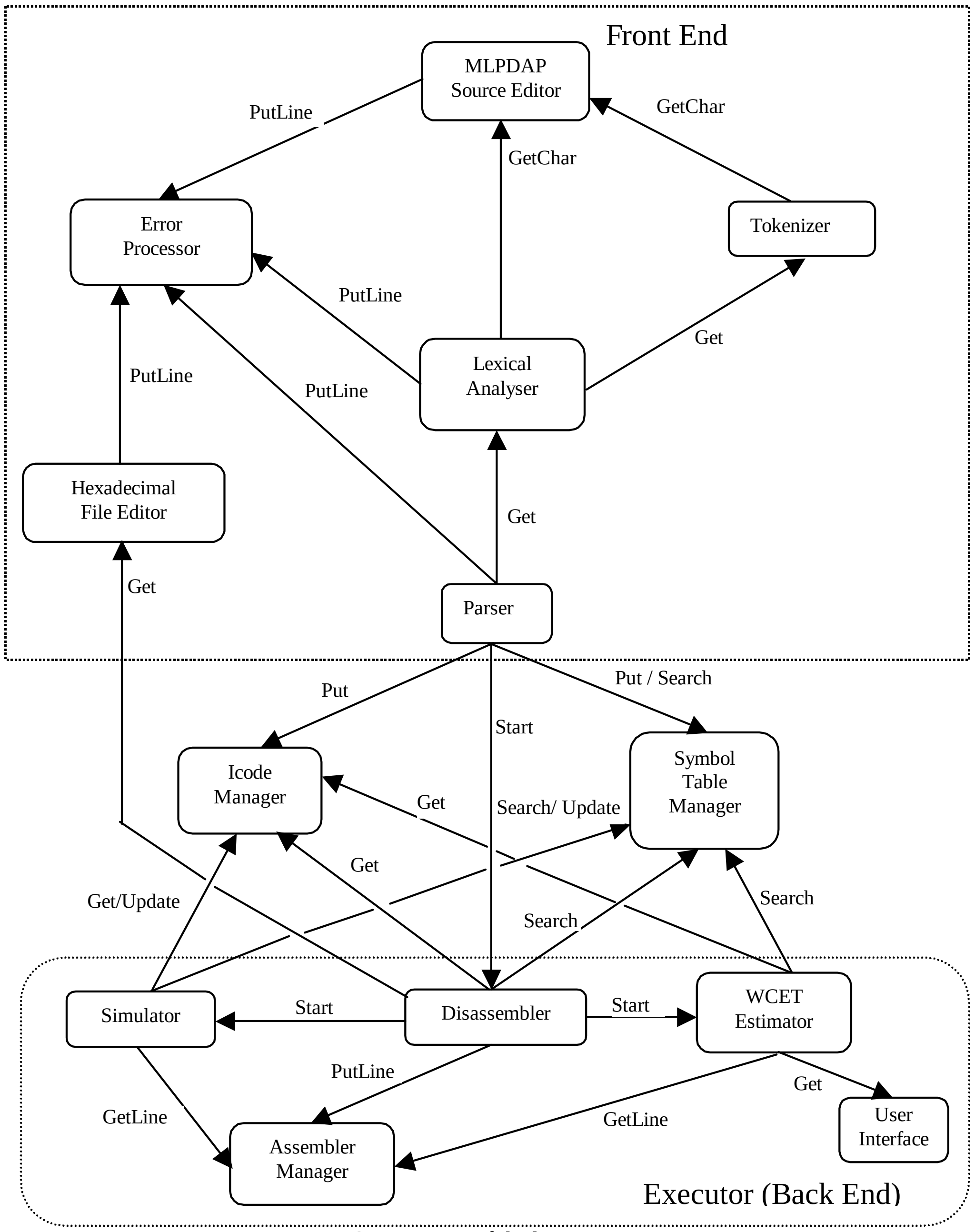
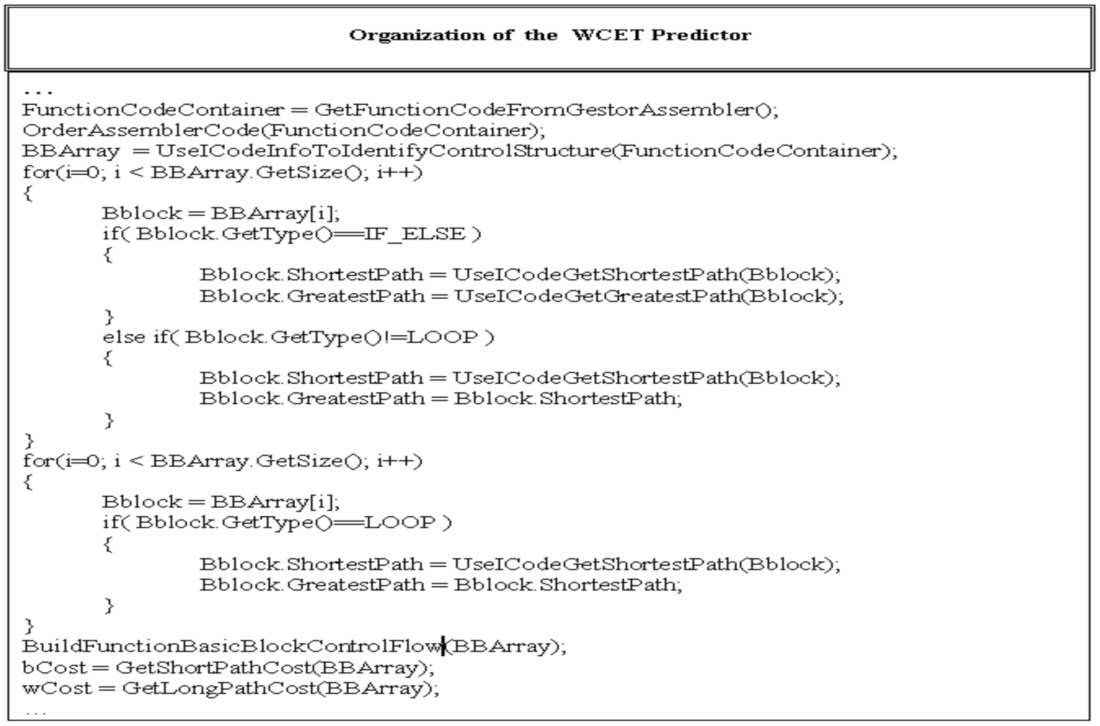


Fig. 1 Organization of the language processor

For our little language, we adopt a procedural and modular paradigm, such that modules are independent from each other, the sequence of modules execution does not matter, but within each module an exact sequence of instructions is specified and the computer executes these instructions in the specified order. A processor's architecture program is written by modules, each one describing a specific feature such as instruction set, interrupt structure and mechanism, register structure, memory organization, pipeline, data cache, instruction cache, PTS, and so on. As said above, the module execution order can be any, but the register module must always be the first to be executed. A module can be defined more than once, but it is a processor language job to verify the information consistency among them and concatenate all them into a single module.

The disassembler has as input an executable file contains the code segment that one wants to measure and the compiled version of the processor's architecture program. The disassembler process starts at the start-up code address (startup code is the bootstrap code executed immediately after the reset or power-on of the processor) and follows the execution flow of the program. It is implemented into four phases:


Organization of the WCET Predictor

```
...
FunctionCodeContainer = GetFunctionCodeFromGestorAssembler();
OrderAssemblerCode(FunctionCodeContainer);
BBArray  = UseICodeInfoToIdentifyControlStructure(FunctionCodeContainer);
for(i=0; i < BBArray.GetSize(); i++)
{
        Bblock = BBArray[i];
        if( Bblock.GetType()==IF_ELSE )
        {
                Bblock.ShortestPath = UseICodeGetShortestPath(Bblock);
                Bblock.GreatestPath = UseICodeGetGreatestPath(Bblock);
        }
        else if( Bblock.GetType()!=LOOP )
        {
                Bblock.ShortestPath = UseICodeGetShortestPath(Bblock);
                Bblock.GreatestPath = Bblock.ShortestPath;
        }
}
for(i=0; i < BBArray.GetSize(); i++)
{
        Bblock = BBArray[i];
        if( Bblock.GetType()==LOOP )
        {
                Bblock.ShortestPath = UseICodeGetShortestPath(Bblock);
                Bblock.GreatestPath = Bblock.ShortestPath;
        }
}
BuildFunctionBasicBlockControlFlow(BBArray);
bCost = GetShortPathCost(BBArray);
wCost = GetLongPathCost(BBArray);
...
```

1. starting at the start-up code address follows all possible execution paths till reaching the end address of the "main" function. At this stage, all function calls are examined and their entry code addresses are pushed into an auxiliary stack,

2. from the entry address of the "main" function, checks the main function code for interrupt activation,

3. for each active interrupt, gets its entry code address and pushed it into the auxiliary stack,

4. pops each entry address from the auxiliary stack and disassemble it, following the function's execution paths.

The execution of the simulation module is optional and the associated process is described by a set of operation introduced using the function "SetAction". That is to say, for each instruction the simulation process, including the flag register affectation, are described by a set of operation specified using "SetAction" calls. To achieve a correct flags affectation, all operations describe by "SetAction" must be implemented using binary base.

Running the simulation process before the estimation process, will produce a more optimistic worst case timing analysis since it can:

a) rectify the execution time of instructions that depend on data locations, such as stack, internal or external memory,

b) solve the indirect address problem by checking if it is a jump or a function call (function call by address),

c) estimate the iteration number of a loop.

The WCET estimator module requires a direct interaction with the user as some parameters are not directly measurable through the program code. Examples of such kind of parameters are, the number of an interrupt occurrence and the preview of a possible maximum iterations number associated to an infinite loop. The WCET estimation process was divided into two phases:

1- first, the code segment to be measured is decomposed into basic blocks,

2- for each basic block, it will be estimated the lower and upper execution time, using the shortest path method and a timing scheme [1].

The shortest path algorithm with the basic block graph as input is used to estimate the lower and upper bound on the execution time of the code segment. For the estimation of the upper bound, it is used the multiplicative inverse of the upper execution time of each basic block.

A basic block is a sequence of assembler's instructions, such as, only the first instruction can be prefixed by a label and only the last one can be a control transfer instruction. The decomposition phase is carried out following the steps below:

1- rearrangement of code segment to guarantee the visual cohesion of a basic block. Note that, the ordering of instructions by address make more difficult the visualization of the inter basic block control flow, due to long jump instructions that can occur between basic blocks. To guarantee that visual cohesion, all sequence of instructions are rearranged by memory address, excluding those one located from long jump labels – these instructions are inserted from the last buffer index.

2- characterization of the conditional structure through the identification of the instructions sequence that compose the "if" and "else" body.

3- characterization of the loop structure through the identification of the instructions sequence that composes the loop body, control and transfer control. It is essential to discern between "while/for" and "do while" loop since the timing schemes are different.

4- After the identification and characterization of the control and loop structures, it will be built a basic block graph, showing all the execution paths between basic blocks.

5- For each basic block, find the lower and upper execution time.

2.1. Pipeline Modelling

The WCET estimator presented so far, considers that an instruction's execution is fixed over the program execution, i.e., it ignores the contribuition of modern processors. Note that, the dependence among instructions can cause pipeline hazards, introducing a delay in the instructions execution. This dependence emerges as several instructions are simultaneously executed and as the result of this parallelism execution among instructions, the execution time of an instruction fluctuates depending on the set of its neighbouring instructions.

Our little language analyses the pipeline using the pipeline hazard detection technique suggested by Proebsting and Fraser [13] and models the pipeline as a set of resources and each instruction as a process that acquires and consumes a subset of resources for its execution. Therefore, the pipeline stages and functional units are defined using functions "setPipeStage(Mn)" and "SetPipeFunctionalUnit(Mn,num)", respectively. For each instruction, there is a set of functions to solve the following points:

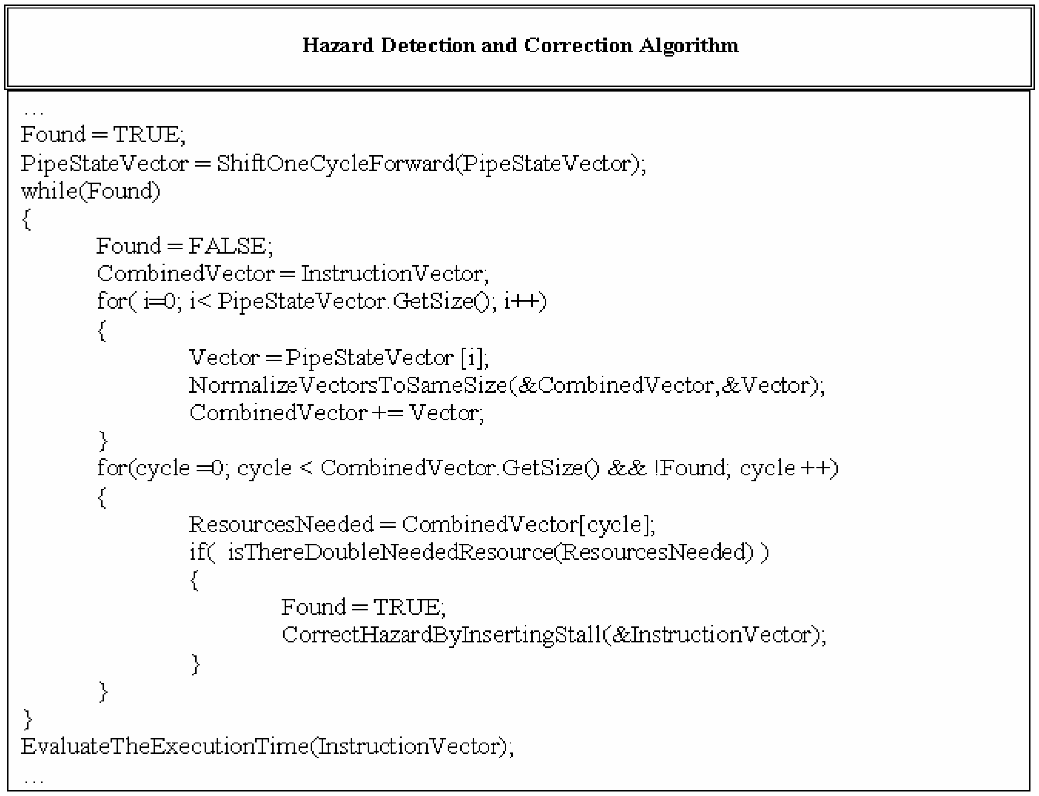
i) Instr.SetSourceStage(s_Opr, Stg) specifies the pipeline stage each source operand must be available,

ii) Instr.SetResultStage(d_Opr, Stg) specifies the pipeline stage the output of the destination operand becomes available.

iii) Instr.SetStageWCET(stg, tm) specifies each pipeline stage required to execute an instruction and the execution time associated to that stage.

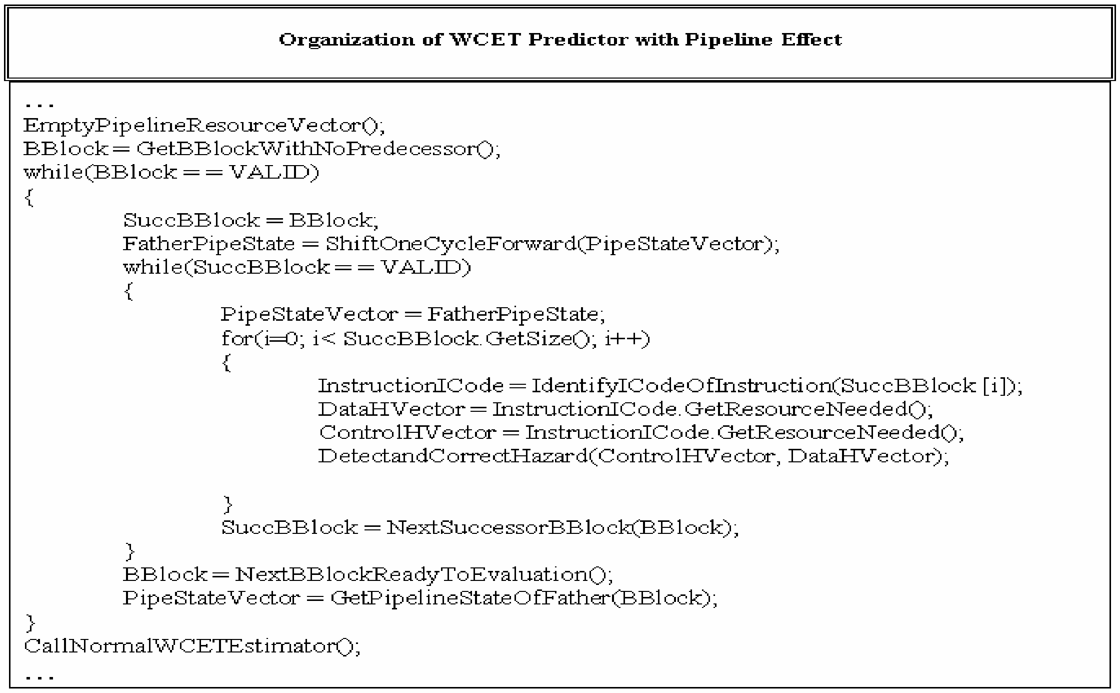
iv) Instr.SetbranchDelayCost(tm) sets the control hazard cost associated to a branch instruction.

The pipeline analysis of a given basic block must always take into account the influences of the predecessor basic blocks (note that, the dependence among instructions can cause pipeline hazards, introducing a delay in the instructions execution), otherwise, it leads to an underestimation of the execution time. Therefore, at the hazard detection stage of a given basic block, it will be always incorporate the pipeline's state associated to the predecessor basic blocks over the execution paths. The resources vector that describes the pipeline's state it will be iteratively updated by inserting pipeline stalls to correct the data and/or structural hazards when the next instruction is issued.



Hazard Detection and Correction Algorithm

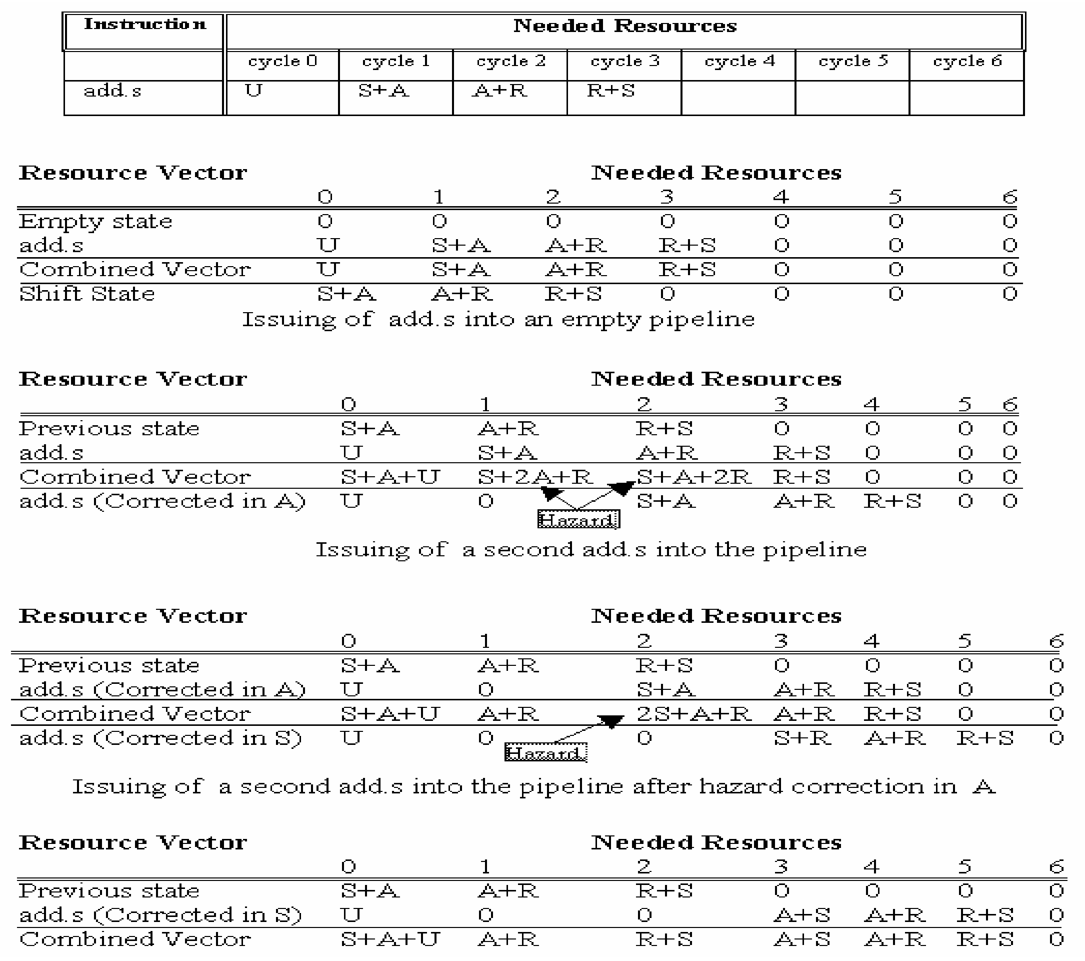
```
...
Found = TRUE;
PipeStateVector = ShiftOneCycleForward(PipeStateVector);
while(Found)
{
        Found = FALSE;
        CombinedVector = InstructionVector;
        for( i=0; i< PipeStateVector.GetSize(); i++)
        {
                Vector = PipeStateVector [i];
                NormalizeVectorsToSameSize(&CombinedVector,&Vector);
                CombinedVector += Vector;
        }
        for(cycle =0; cycle < CombinedVector.GetSize() && !Found; cycle ++)
        {
                ResourcesNeeded = CombinedVector[cycle];
                if(  isThereDoubleNeededResource(ResourcesNeeded) )
                {
                        Found = TRUE;
                        CorrectHazardByInsertingStall(&InstructionVector);
                }
        }
}
EvaluateTheExecutionTime(InstructionVector);
...
```



Organization of WCET Predictor with Pipeline Effect

```
...
EmptyPipelineResourceVector();
BBlock = GetBBlockWithNoPredecessor();
while(BBlock = = VALID)
{
        SuccBBlock = BBlock;
        FatherPipeState = ShiftOneCycleForward(PipeStateVector);
        while(SuccBBlock = = VALID)
        {
                PipeStateVector = FatherPipeState;
                for(i=0; i< SuccBBlock.GetSize(); i++)
                {
                        InstructionICode = IdentifyICodeOfInstruction(SuccBBlock [i]);
                        DataHVector = InstructionICode.GetResourceNeeded();
                        ControlHVector = InstructionICode.GetResourceNeeded();
                        DetectandCorrectHazard(ControlHVector, DataHVector);

                }
                SuccBBlock = NextSuccessorBBlock(BBlock);
        }
        BBlock = NextBBlockReadyToEvaluation();
        PipeStateVector = GetPipelineStateOfFather(BBlock);
}
CallNormalWCETEstimator();
...
```



Instruction	Needed Resources						
	cycle 0	cycle 1	cycle 2	cycle 3	cycle 4	cycle 5	cycle 6
add.s	U	S+A	A+R	R+S			

Resource Vector	Needed Resources						
	0	1	2	3	4	5	6
Empty state	0	0	0	0	0	0	0
add.s	U	S+A	A+R	R+S	0	0	0
Combined Vector	U	S+A	A+R	R+S	0	0	0
Shift State	S+A	A+R	R+S	0	0	0	0

Issuing of add.s into an empty pipeline

Resource Vector	Needed Resources						
	0	1	2	3	4	5	6
Previous state	S+A	A+R	R+S	0	0	0	0
add.s	U	S+A	A+R	R+S	0	0	0
Combined Vector	S+A+U	S+2A+R	S+A+2R	R+S	0	0	0
add.s (Corrected in A)	U	0	S+A	A+R	R+S	0	0

Issuing of a second add.s into the pipeline

Resource Vector	Needed Resources						
	0	1	2	3	4	5	6
Previous state	S+A	A+R	R+S	0	0	0	0
add.s (Corrected in A)	U	0	S+A	A+R	R+S	0	0
Combined Vector	S+A+U	A+R	2S+A+R	A+R	R+S	0	0
add.s (Corrected in S)	U	0	0	S+R	A+R	R+S	0

Issuing of a second add.s into the pipeline after hazard correction in A

Resource Vector	Needed Resources						
	0	1	2	3	4	5	6
Previous state	S+A	A+R	R+S	0	0	0	0
add.s (Corrected in S)	U	0	0	A+S	A+R	R+S	0
Combined Vector	S+A+U	A+R	R+S	A+S	A+R	R+S	0

Fig.2 Illustration of the hazard detection and correction

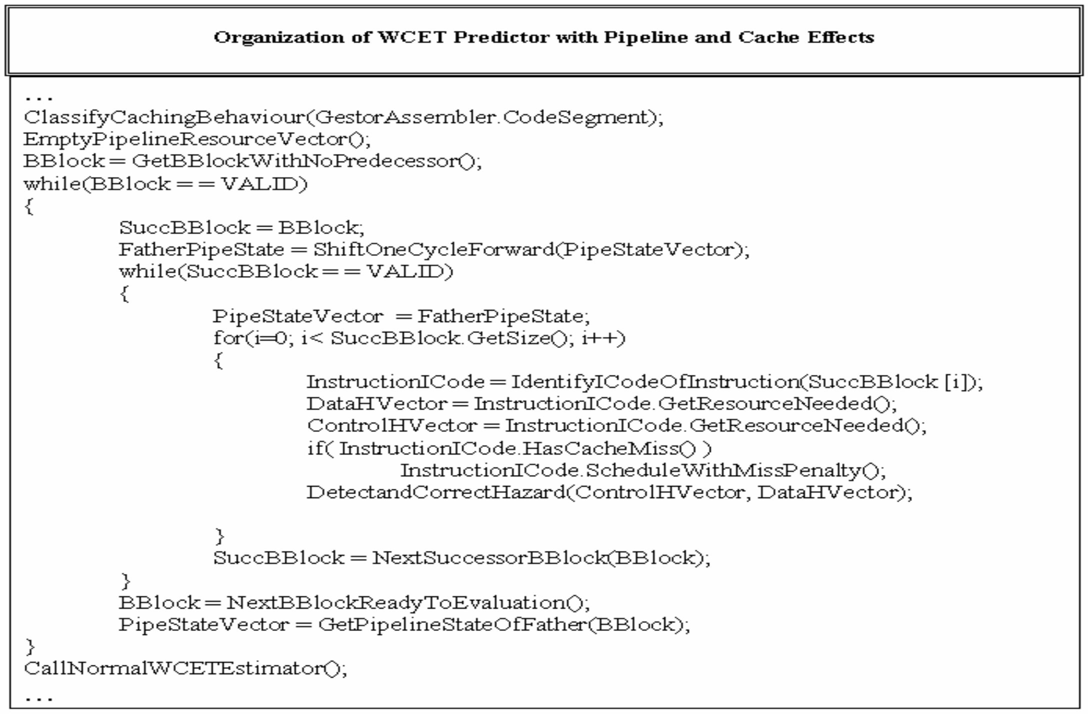
If these two hazards happen simultaneously, the correction process start at the hazard that occurred first and after it is checked if the second one still remains. The issuing of the new instruction will be always preceded by the updating of the previous pipeline's state. This is achieved by shifting the actual pipeline resource vector one cycle forward.

The pipeline architectures, usually, present special techniques to correct the execution flow when a control hazard happens. For instance, the delay transfer control technique offers the hardware an extra machine cycle to decide the branch. Also, special hardware is used to determine the branch label and value condition at the end of the instruction's decode. As one can conclude, the execution of delay instructions does not depend on the branch decision and it is always carried out. So, we model the control hazard, as being caused by all kind of branch instruction and by adding the sum of execution time of all instruction in the slot delay to the basic block execution time.

2.2. Cache Modelling

Cache is a high speed and small size memory, typically, a SRAM that contains parts of the most recent accesses to the main memory. Nowadays, the time necessary to load an instruction or data to the processor is much longer than the instruction execution time. The main rule of a cache memory is to reduce the time needed to move the information from and to the processor. An explanation for this improvement, comes from the locality of reference theory – at any time, the processor will access a very small and localized region of the main memory and the cache load this region, allowing faster memory accesses to the processor.

In spite of the memory performance enhancement, the cache makes the execution time estimation harder, as the execution time of any instruction will vary and depend on the presence of the instruction and data into the caches. Furthermore, to exactly know if the execution of a given instruction causes a cache miss/hit, it will be necessary to carry out a global analysis of the program. Note that an instruction's behaviour can be affected by memory references that happened long time before.



Organization of WCET Predictor with Pipeline and Cache Effects

```
...
ClassifyCachingBehaviour(GestorAssembler.CodeSegment);
EmptyPipelineResourceVector();
BBlock = GetBBlockWithNoPredecessor();
while(BBlock = = VALID)
{
        SuccBBlock = BBlock;
        FatherPipeState = ShiftOneCycleForward(PipeStateVector);
        while(SuccBBlock = = VALID)
        {
                PipeStateVector  = FatherPipeState;
                for(i=0; i< SuccBBlock.GetSize(); i++)
                {
                        InstructionICode = IdentifyICodeOfInstruction(SuccBBlock [i]);
                        DataHVector = InstructionICode.GetResourceNeeded();
                        ControlHVector = InstructionICode.GetResourceNeeded();
                        if( InstructionICode.HasCacheMiss() )
                                InstructionICode.ScheduleWithMissPenalty();
                        DetectandCorrectHazard(ControlHVector, DataHVector);

                }
                SuccBBlock = NextSuccessorBBlock(BBlock);
        }
        BBlock = NextBBlockReadyToEvaluation();
        PipeStateVector = GetPipelineStateOfFather(BBlock);
}
CallNormalWCETEstimator();
...
```

Adversely, the estimation of WCET becomes harder for the modern processors, as the behaviour of cache and pipeline depend on each other. Therefore, we propose the following changes to the algorithm that takes into account the pipeline effects:

1) Classify the cache behaviour [9] for any data and instruction as cache hit or cache miss before the analysis of the pipeline behaviour

2) Before the issuing of an instruction, verify if there is any cache miss related to the instruction, and if any, apply the miss penalty beforehand and then the detection and correction of pipeline hazards

3. EXPERIMENTAL RESULTS

By the moment, we will present some results using the 8xC196 Intel microcontrollers as they are the only that presents all needed execution time information in the user's guide. But we hope soon to present results of experiments with modern processor such as, some Texas Instruments DSPs, Intel 8xC296, PICs and so on.

Fig.2 shows the program to be estimated, that is composed by two functions: the *main()* and *func()*. This program was instrumented to allow a direct measurement with a digital oscilloscope through the pin number six of port 2 (P2.6).

At a first stage, the WCET estimator built the call graph

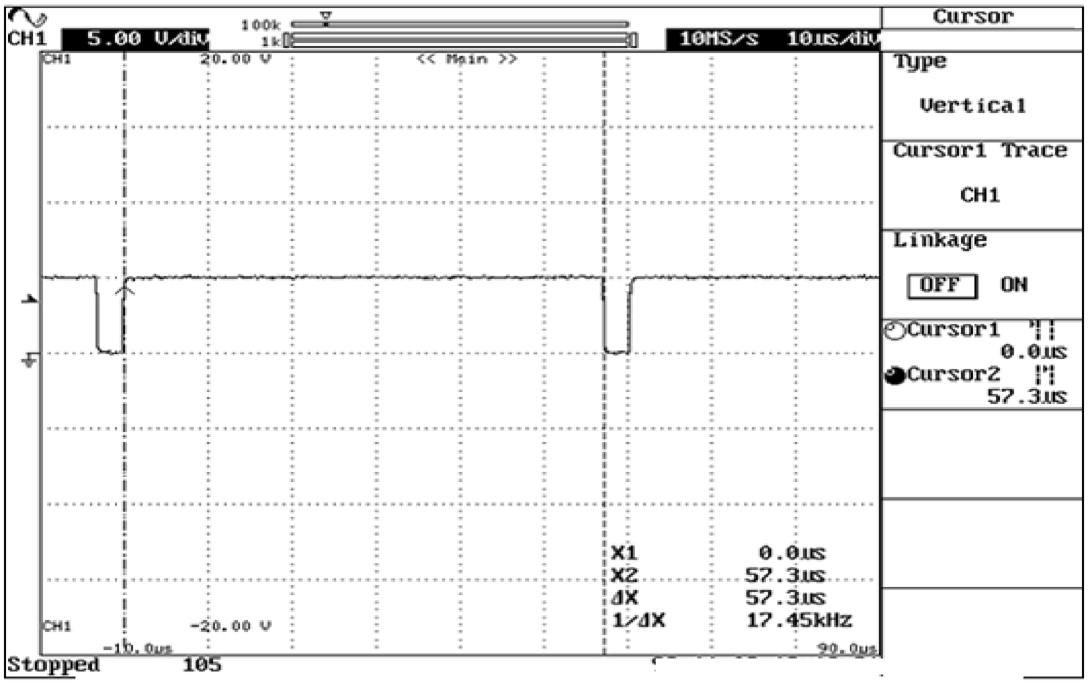
```
#pragma model(MC)
#include        _SFR_H_
#include        _FUNCS_H_
/*   Reserve the 9 bytes required by eval board   */
char                    reserve[9];
#pragma locate(reserve=0x30)
register int i,k;
register int x, y;
void func();

void main(void)
{
        x = 1;
        y = 0;
        i = 20;
        func();
}
```

```
void func()
{
        for(k=1; k<10; k++)
        {
                x*=3;
                y = x + 4;
        }
}
```

Fig.2 Code segment to be measured

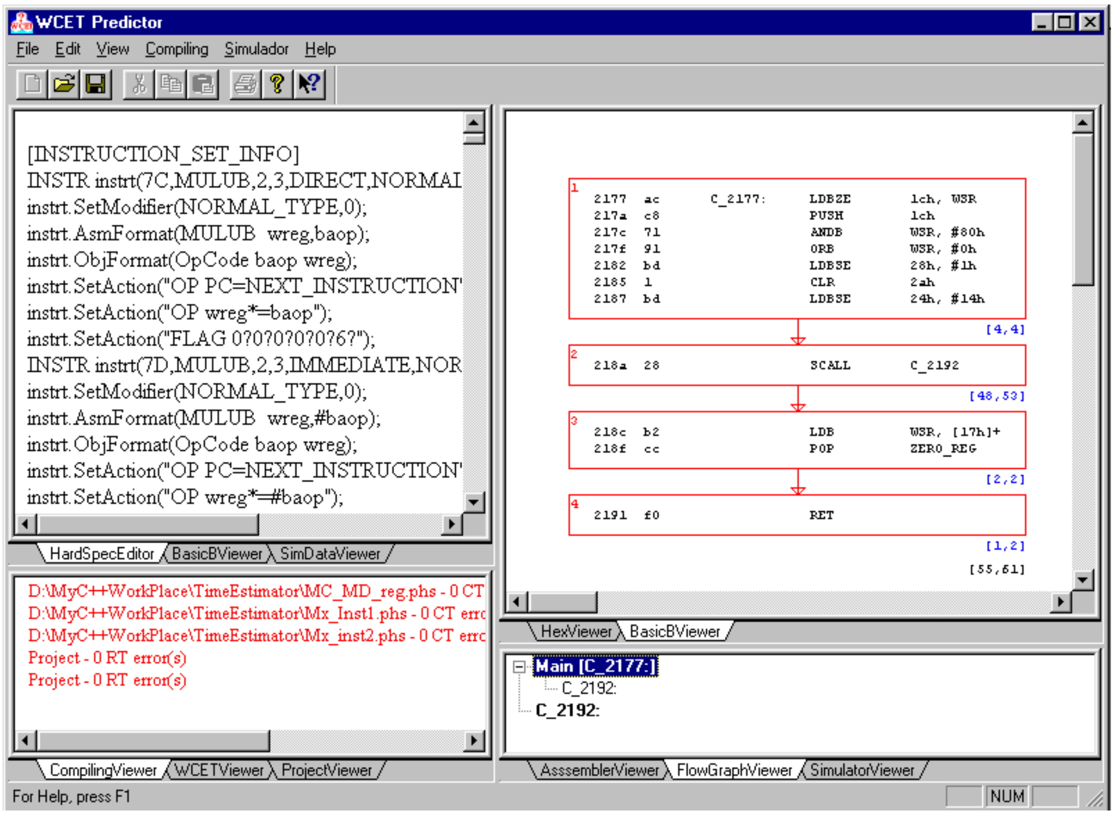
given at the lower right quadrant of fig.5 and then, *func()* identified by the label C_2192 will be processed and the result shown by the fig.4. At the upper right quadrant of fig.4 and 5, information such as execution time of individual basic blocks, basic block control flow and function execution time are presented. At the lower right quadrant of fig.5, is presented the assembler code translated by the disassembler from the executable code. The upper left quadrant of fig.4 and 5 present parts our litlle language program describing the microcontroller architecture.



```
Cursor
Type
  Vertical
Cursor1 Trace
  CH1
Linkage
  OFF  ON
Cursor1
  0.0µs
Cursor2
  57.3µs
```

```
X1       0.0µs
X2      57.3µs
dX      57.3µs
1/dX    17.45kHz
```

Fig.3 Direct measurement of Instrumented program using a digital Oscilloscope to monitor P2.6



WCET Predictor
File Edit View Compiling Simulador Help
HardSpecEditor BasicBViewer SimDataViewer
HexViewer BasicBViewer
CompilingViewer WCETViewer ProjectViewer
AsssemblerViewer FlowGraphViewer SimulatorViewer
For Help, press F1
NUM

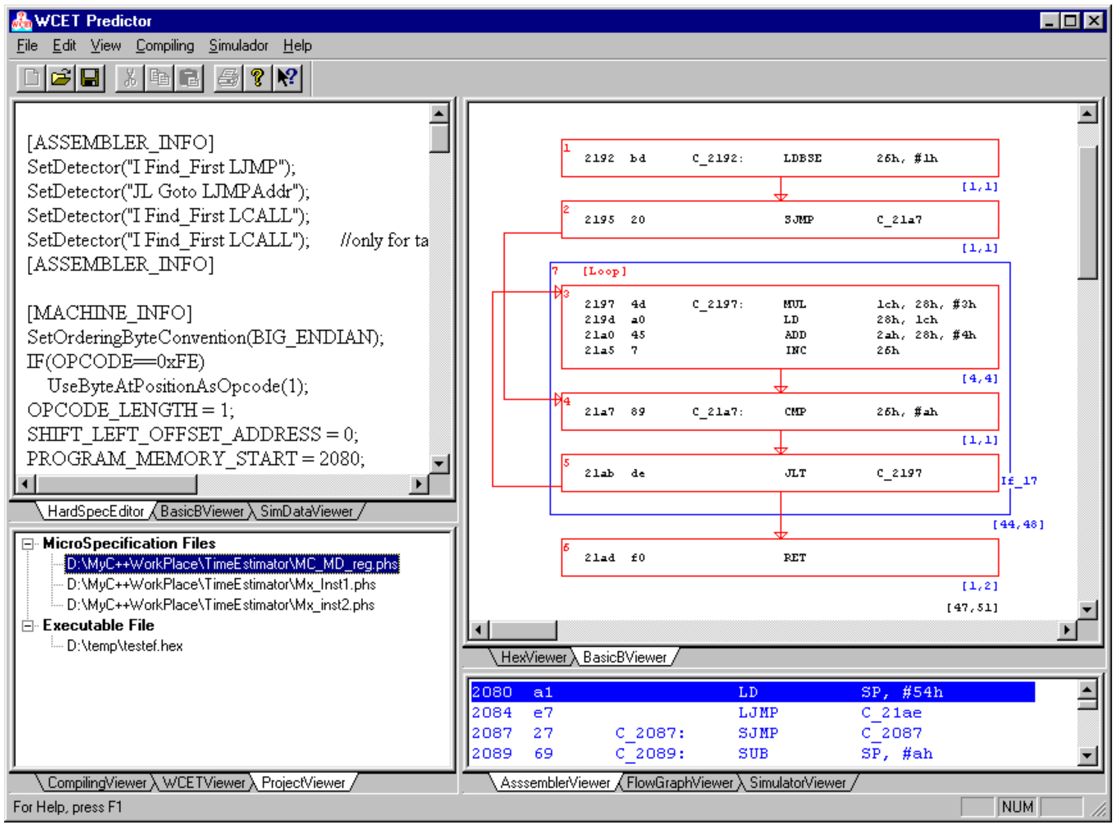
```
[INSTRUCTION_SET_INFO]
INSTR instr(7C,MULUB,2,3,DIRECT,NORMAI
instr.SetModifier(NORMAL_TYPE,0);
instr.AsmFormat(MULUB  wreg,baop);
instr.ObjFormat(OpCode baop wreg);
instr.SetAction("OP PC=NEXT_INSTRUCTION'
instr.SetAction("OP wreg*=baop");
instr.SetAction("FLAG 0?0?0?0?0?6?");
INSTR instr(7D,MULUB,2,3,IMMEDIATE,NOR
instr.SetModifier(NORMAL_TYPE,0);
instr.AsmFormat(MULUB  wreg,#baop);
instr.ObjFormat(OpCode baop wreg);
instr.SetAction("OP PC=NEXT_INSTRUCTION'
instr.SetAction("OP wreg*=#baop");
```

```
D:\MyC++WorkPlace\TimeEstimator\MC_MD_reg.phs - 0 CT
D:\MyC++WorkPlace\TimeEstimator\Mx_Inst1.phs - 0 CT erro
D:\MyC++WorkPlace\TimeEstimator\Mx_inst2.phs - 0 CT erro
Project - 0 RT error(s)
Project - 0 RT error(s)
```

```
1
 2177  ac    C_2177:   LDBZE    1ch, WSR
 217a  c8              PUSH     1ch
 217c  71              ANDB     WSR, #80h
 217f  91              ORB      WSR, #0h
 2182  bd              LDBSE    28h, #1h
 2185  1               CLR      2ah
 2187  bd              LDBSE    24h, #14h
                                          [4,4]
2
 218a  28              SCALL    C_2192
                                          [48,53]
3
 218c  b2              LDB      WSR, [17h]+
 218f  cc              POP      ZERO_REG
                                          [2,2]
4
 2191  f0              RET
                                          [1,2]
                                          [55,61]
```

```
Main [C_2177:]
   C_2192:
C_2192:
```

Fig. 4 WCET = 61µs was estimated for the code presented at fig. 2



WCET Predictor
File Edit View Compiling Simulador Help
HardSpecEditor BasicBViewer SimDataViewer
CompilingViewer WCETViewer ProjectViewer
HexViewer BasicBViewer
AsssemblerViewer FlowGraphViewer SimulatorViewer
For Help, press F1
NUM

```
[ASSEMBLER_INFO]
SetDetector("I Find_First LJMP");
SetDetector("JL Goto LJMPAddr");
SetDetector("I Find_First LCALL");
SetDetector("I Find_First LCALL");       //only for ta
[ASSEMBLER_INFO]

[MACHINE_INFO]
SetOrderingByteConvention(BIG_ENDIAN);
IF(OPCODE==0xFE)
   UseByteAtPositionAsOpcode(1);
OPCODE_LENGTH = 1;
SHIFT_LEFT_OFFSET_ADDRESS = 0;
PROGRAM_MEMORY_START = 2080;
```

```
MicroSpecification Files
    D:\MyC++WorkPlace\TimeEstimator\MC_MD_reg.phs
    D:\MyC++WorkPlace\TimeEstimator\Mx_Inst1.phs
    D:\MyC++WorkPlace\TimeEstimator\Mx_inst2.phs
Executable File
    D:\temp\testef.hex
```

```
1
  2192  bd   C_2192:   LDBSE   26h, #1h
                                          [1,1]
2
  2195  20             SJMP    C_21a7
                                          [1,1]
7   [Loop]
3
  2197  4d   C_2197:   MUL     1ch, 28h, #3h
  219d  a0             LD      28h, 1ch
  21a0  45             ADD     2ah, 28h, #4h
  21a5  7              INC     26h
                                          [4,4]
4
  21a7  89   C_21a7:   CMP     26h, #ah
                                          [1,1]
5
  21ab  de             JLT     C_2197          If_17
                                              [44,48]
6
  21ad  f0             RET
                                          [1,2]
                                         [47,51]
```

```
2080  a1              LD      SP, #54h
2084  e7              LJMP    C_21ae
2087  27    C_2087:   SJMP    C_2087
2089  69    C_2089:   SUB     SP, #ah
```

Fig. 5 WCET analysis of the funtion denominated func()

4. CONCLUSIONS

A very friendly tool for the WCET estimation was developed and the results obtained over some Intel microcontroller were very satisfactory. To a complete evaluation of our tool we will realize more test using other classes of processors such as DSPs, PICs and some Motorola microcontrolers.

A plenty use of this tool requires some processors informations, such as, the execution time of each instructions composing the processor instruction set, sometimes not provided in the processor user's guide. In such case, to time an individual instruction, we recommended the use of the logic analyzer following the next four steps:

1. write a short program that contains the target instruction

2. find the memory location in the code segment of memory containing the target instruction

3. set the logic analyzer to trigger on the opcode at the target instruction location and on the opcode and location of the next instruction

4. set the trace for absolute time

and the logic analyzer will then display the difference in the time between the fetch of the target and next instructions.

As a future works, we propose the acceleration of convergency for some of the implemented scheduling methods, support of interprocessor topology based on I^2C and CAN. As a final work is our intent to grant this tool with a full round-trip engineering based on reverse engineering feature that quickly informs the effects of a graphical change over the application design.

5. REFERENCES

[1] Alan C. Shaw, Deterministic Timing Schema for Parallel Programs, technical Report 90-05-06, Department of Computer Science and Engineering, University of Washington, Seattle, 1990.

[2] P. Puschner and CH. Koza, Calculating the Maximum Execution Time of Real-Time Programs, The Journal of Real-Time Systems, 1, pp. 159-176,1989.

[3] A. K. Mok et al., Evaluating Tight Execution Time Bounds of Programs by Annotations, in Proc. Of the Sixth IEEE Workshop on Real-Time Operating Systems and Software, pp. 272-279, May 1989.

[4] S. Bharrat and K. Jeffay, Predicting Worst case Execution Times on a Pipelined RISC Processor, Technical Report, Department of Computer Science, University of North Carolina.

[5] K. Nilsen and B. Rygg, Worst-Case Execution Time Analysis on Modern Processor, ACM SIGPLAN Notices, Vol. 30, No. 11, pp. 20-30, November 1995.

[6] Y. Steven Li et al., Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software, Technical Report, Department of Electrical Engineering, Princeton University.

[7] N. Zhang and M. Nicholson, Pipelined Processors and Worst Case execution Times, Real-Time Systems Journal, Vol. 5, No. 4, pp. 319-343, October 1993.

[8] Tai-Yi Huang et al., A Method for Bounding the Effect of DMA I/O Interference on Program Execution Time, in Proc. Real-Time Systems Symposium, Washington DC. December 1996.

[9] C. Healy, D. Whalley and M. Harmon, Integrating the Timing Analysis of Pipelining and Instruction Caching, Technical Report, Computer Science Department, Florida State University.

[10] Sung-Soo Lim, C. Yun Park et al., An Accurate Worst Case Timing Analysis for RISC Processors, IEEE Transactions on Software Engineering, Vol. 21, No. 7, pp. 593 – 604, July 1995.

[11] Eric Schnarr and James Larus, Instruction Scheduling and Executable Editing, Worshop on Compiler Support for System Software (WCSSS' 96), Tucson, Arizona, February, 1996.

[12] J. Tremblay and P. Sorenson, The Theory and Practice of Compiler Writing, McGraw-Hill, ISBN 0-07-065161-2, 1987.

[13] T. Proebsting and C. W. Fraser, Detecting Pipeline Structural Hazards Quickly, in Proc. of the 21th Annual ACM SIGPLAN_SIGACT Symposium on Principles of Programming Languages, pp. 280-286, January 1994.