Lab Assignments
Advanced Computer Architecture

28. April 2015

This document contains the four lab assignments of the Advanced Computer Architectures (ACA) course. The four assignments are the WinMIPS64, SIMD, SimpleScalar and the SMT assignment. It is recommended that you do the assignments in their respective order so that the lab and lecture topics are synchronized.

Each assignments has deliverables which are further described in the assignments. The deliverables have to be submitted via the ISIS page of ACA, which has a link for each assignment in the Grading and Examination section. The assignments can be performed in groups of two students. Submit only one assignment per group.

Deadlines of the assignments are:

WinMIPS64	**18.05.2015**
SIMD	**08.06.2015**
SimpleScalar	**22.06.2015**
SMT	**06.07.2015**

Submitting the deliverables past the deadline reduces your maximum points for the respective assignment by 5 points each day. You can submit the assignments until midnight on the day of the deadline.

1. Fachgebiet Architektur eingebetteter Systeme
Advanced Computer Architectures SS2015

Assignment start: 28.04.2015 Submission deadline: 18.05.2015

Assignment 1 - WinMIPS64 25 Points

The first assignment of the lab for Advanced Computer Architecture (ACA) is the WinMIPS64 assignment. In this assignment you will play the role of a C to MIPS64 assembly compiler. The WinMIPS64 simulator, which is a simulator of a 64-bit 5-stage pipelined MIPS architecture, is be used to verify correctness and evaluate performance.

The simulator, documentation, and the to be converted C-program `smooth.c` are available at *https://www.isis.tu-berlin.de/2.0*. It is highly recommended that you read the documentation and experiment with the example programs before starting the real C to Mips64 conversion process.

Use the labels of Listing 1 at the start of your assembly file as the variables. In our testing procedure we will use the same labels, but with different contents, for example, as in Listing 2. The content of `coeff` and `sample` should, therefore, not be assumed static. The label(`N_SAMPLES`) should be changed according to the number of declared samples. The `sample` and `result` will always have the same number of samples. Also the number of coefficients (`N_COEFFS`) is fixed at 3 for all test sequences.

Listing 1: MIPS64 program heading

```
.data
N_COEFFS:    .word    3
coeff:       .double  0.5,1.0,0.5
N_SAMPLES:   .word    5
sample:      .double  1.0,2.0,1.0,2.0,1.0
result:      .double  0.0,0.0,0.0,0.0,0.0
```

Listing 2: Possible testing sequence

```
N_COEFFS     .word    3
coeff:       .double  0.5,1.0,0.5
N_SAMPLES:   .word    10
sample:      .double  1,2,3,4,5,6,7,8,9,10
result:      .double  0,0,0,0,0,0,0,0,0,0
```

The MIPS64 code in Listing 3 can be used to print a value. This is useful to replace `printf` calls to confirm your output while programming. The `printf` calls in `smooth.c` do not have to be converted and calls to `printdouble` should be removed in the final assembly file. They are there for debugging purposes. Alternatively, you can also look at the register contents when stepping through the instructions.

The WinMIPS64 simulator has to be configured to have a floating point addition delay of **3** cycles, floating point multiplication of **4** cycles, and a division delay of **12** cycles. Also configure WinMIPS64 to use **“data forwarding"** and the **“branch delay slot"**. The grading for this assignment is 10 points

Listing 3: Print doubles

```
printdouble:
    lwu r11,CR(r0)  ; Control Register
    lwu r12,DR(r0)  ; Data Register
    daddi   r10,r0,3    ; r10 = 3
    s.d f0,(r12)    ; output f0 ...
    sd  r10,(r11)   ; ... to screen
    jr  r31         ; return
```

for functional correctness of your assembly program. The remainder of points are earned based on your execution time. The faster you are the more points you earn. A requirement is that the code size reported by WinMIPS64 may not exceed 350 bytes. If your code has too many bugs, i.e, it cannot be fairly compared in terms of speed, you will not receive any points for execution time. The performance tests will be performed with our non-disclosed testing sequences.

A few tips:

- Optimize the C-program first before converting to MIPS64. This will decrease conversion complexity and improve performance significantly.
- Check the correctness after each step you perform.
- Commit or backup your work frequently, so you can easily revert to an earlier version when something goes wrong.
- Write comments for your own convenience. Content of registers is mostly useful information.
- We will test with sequences of various sizes, small (<10 elements) to large (between 40 and 100). Your performance result is the combined number of cycles of the test sequences.
- The instruction set syntax used in WinMIPS64 is slightly different from what is used in the lecture slides. Code copied from the slides will not work. Read the documentation and/or look at the example programs to familiarize yourself with the WinMIPS64 instruction syntax.

The deliverable of this assignment is your assembly file of the `smooth` program. Do not include any calls to `printdouble` or the function itself in the assembly file you submit. Also please refrain from changing the label names in Listing 1 as it will help in our checking procedure. New labels can be added but should not replace the old ones.

2.	Fachgebiet Architektur eingebetteter Systeme **Advanced Computer Architectures**	SS2015

Assignment start: 19.05.2015 Submission deadline: 08.06.2015

Assignment 2 - SIMD 25 Points

In the second assignment you will use x86 SIMD extensions to accelerate the HEVC/H.265 16×16 inverse transform. Single Instruction Multiple Data instructions are useful in many applications to improve the performance. On modern processors these instructions are always available in sometimes many extension sets. For x86 there are the MMX (64-bit), SSE, SSE2, SSSE3, SSE4.1, SSE4.2 (128-bit), and recently AVX (256-bit) extensions available. In the near future AVX2 and FMA3/4 will be added. For conventional C statements, however, these instructions are typically left unused by the compiler. Sometimes, the compiler can auto-vectorize certain loops, but at the moment rarely generate satisfactory code. To use these instruction programmers usually write either direct assembly (gcc inline asm, yasm, nasm) or intrinsics.

In the lab you will use intrinsics, which provide a more convenient interface for using these SIMD instructions to programmers. The programmer still has to perform the so called vectorization, but is relieved from performing register allocation and spilling. An example of using SIMD instructions using intrinsics is provided in Listing 4 that performs a square and an add on each element of an array of 100 32-bit integers using SSE*.

Listing 4: Example of using x86 SSE intrinsics

```
int input[100]  __attribute__ ((aligned (16)));
int output[100] __attribute__ ((aligned (16)));

//scalar code
for (int i=0; i<100; i++){
  output[i] = input[i]*input[i] + 3;
}

//intrinsics code
__m128i *in_vec = (__m128i *) input;          //cast to SIMD vector type
__m128i *out_vec = (__m128i *) output;
__m128i xmm0;
__m128i add3 = _mm_set1_epi32(3);             //set a vector to (0
    x0003000300030003)
for (int i=0; i<25; i++){
  xmm0 = _mm_load_si128 (&in_vec[i]);         //load 4 integers
  xmm0 = _mm_mullo_epi32(xmm0, xmm0);         //square 4 integers
  xmm0 = _mm_add_epi32  (xmm0, add3);         //add 4 integers
  _mm_store_si128(&out_vec[i], xmm0);         //store back 4 integers
}
```

And another example in Listing 5 which transposes a 4x4 integer matrix.

The inverse discrete cosine transform (IDCT) used in this assignment is based on the fast 16 × 16 transform of the HEVC/H.265 video codec. This 2D IDCT can be decomposed in a 1D row and column IDCT, which is employed in the provided scalar implementation. The scalar code along with other documents can be downloaded from the ISIS page. Your assignment is to create the SIMD

Listing 5: Transposing a 4x4 matrix with SSE

```c
int matrix[4][4] __attribute__ ((aligned (16)));

//intrinsics code
__m128i *matrix_vec = (__m128i *) matrix;

__m128i I0 = _mm_load_si128 (&matrix_vec[0]);    //load 4x4 matrix
__m128i I1 = _mm_load_si128 (&matrix_vec[1]);
__m128i I2 = _mm_load_si128 (&matrix_vec[2]);
__m128i I3 = _mm_load_si128 (&matrix_vec[3]);

__m128i T0 = _mm_unpacklo_epi32(I0, I1);
__m128i T1 = _mm_unpacklo_epi32(I2, I3);
__m128i T2 = _mm_unpackhi_epi32(I0, I1);
__m128i T3 = _mm_unpackhi_epi32(I2, I3);

I0 = _mm_unpacklo_epi64(T0, T1);            //Assigning transposed values back
    into I[0-3]
I1 = _mm_unpackhi_epi64(T0, T1);
I2 = _mm_unpacklo_epi64(T2, T3);
I3 = _mm_unpackhi_epi64(T2, T3);

_mm_store_si128 (&matrix_vec[0], I0);          //store transposed 4x4 matrix
_mm_store_si128 (&matrix_vec[1], I1);
_mm_store_si128 (&matrix_vec[2], I2);
_mm_store_si128 (&matrix_vec[3], I3);
```

intrinsics version of the IDCT. The correctness and timing checking functionality is also available in the source file .

A few tips and rules:

- Intel has a nice tool, the “Intel intrinsics guide”. This tool contains compact and convenient documentation of all the available SIMD intrinsics for x86. A recent version is in the assignment sources on ISIS. The latest version can also be downloaded at `http://software.intel.com/en-us/avx/`.**Try this tool before doing anything else.**
- Use only intrinsics up to SSE4.2. AVX and 3-operand mode SSE are not supported on the lab machines.
- Generating the input in a transposed manner is allowed.
- Do not unroll the timing loop.
- Do not inline the idct functions.
- Variations in execution time are normal, since the thin clients in the lab room connect to virtual machines. When evaluating the performance, run the code multiple times and consider the fastest time as representative.
- You can use `objdump` to see the generated assembly code.
- If you are not satisfied with the code generation of intrinsics you are allowed to use GCC inline assembly instead.
- When working from another machine it is necessary to check the support for newer instructions. For instance, support for SSE4.2 intrinsics on older machines could be missing.

The deliverable of this assignment is the *hevc_idct16.c* source file.

3. Fachgebiet Architektur eingebetteter Systeme
Advanced Computer Architectures SS2015

Assignment start: 09.06.2015 Submission deadline: 22.06.2015

Assignment 3 - SimpleScalar 25 Points

In the third assignment a more advanced processor simulator is used, SimpleScalar. You will investigate the IPC improvements resulting from key advances in processor microarchitecture. The SimpleScalar simulator can simulate (single-core) processors with a variety of configurations. The type of branch predictor, number of issue slots, cache size, cache latency, and other architectural parameters can be freely configured. Documentation about the SimpleScalar simulator is available at `http://www.simplescalar.com/docs/users_guide_v2.pdf`. In this assignment we only use the *sim-outorder* version of the simulator. The simulator only runs on Linux (x86, 64-bit).

Five precompiled benchmark programs for the SimpleScalar simulator are available. The simulator and the benchmarks can be found in */afs/tu-berlin.de/units/Fak_IV/aes/aca/SimpleScalar* and archived on the ISIS page.

- *Fibonacci* - calculates the 20th instance of Fibonacci sequence
- *Matmul* - matrix multiplication of size 50x50
- *Pi* - estimates the pi number
- *Whetstone* - a double precision benchmark
- *Memcopy* - a memory benchmark

In the first four exercises you have to produce bar graphs of the effect of improving a single feature. Put the benchmarks on the horizontal axis, and the IPC on the vertical axis.

1. (5 Points) Investigate the effect of the branch predicting modes offered by SimpleScalar. You can exclude the *taken* predictor option from the analysis. Due to a simulator bug it produces the same results as *nottaken*. Set the other parameters to configure the simulated processor as a single-issue, in-order processor, with one integer and floating point alu and mult unit. Manipulate the -fetch:ifqsize, -decode:width, -issue:width, -commit:width, -issue:inorder, -res:ialu, -res:imult, -res:fpalu and -res:fpmult options to achieve this. The number of memports should stay at the default value of 2.

   Perform the experiment for each benchmark and put the IPC results in a bar graph. For each benchmark, the result of the different branch predictors must be clustered in the following order: nottaken, bimod, 2lev, comb, perfect.

2. (5 Points) Investigate the effect of increasing the fetch, decode, issue, commit width and the number of resources (“-res:” options) for the options 1, 2, 4 and 8. Also investigate a fifth option with a processor width of 8, but using default amount of resources (4 ialu, 1 imult, 4 fpalu, 1 imult). The number of memports should stay at the default value of 2. Set the simulator to use the 2lev branch predictor.

   Perform the experiment for both in-order and out-of-order execution. This should result in two bar graphs with each five clustered bars for each benchmark with increasing processor width. Put the fifth experiment at the end of each cluster.

3. (5 Points) Investigate the effect of increasing the size of the register update unit (ruu) and load store queue (lsq). Set the processor width to 8 and use out-of-order execution (manipulate the

-fetch:ifqsize, -decode:width, -issue:width, -commit:width and -issue:inorder options). Use the combinations (ruu, lsq) of (16, 8), (32, 8), (32, 16), (64, 16) and (64, 32).The number of memports should stay at the default value of 2. The simulator should use the 2lev branch predictor.

Investigate this for the default amount of execution resources (4/1/4/1) and for a doubled amount (8/2/8/2), resulting in two bar graphs.

4. (5 Points) Write a small report (350 words) on your findings in the previous assignments. Explain what limits the IPC for the benchmarks initially and how branch prediction, processor width, and increasing the instruction window and load store queue overcome this. Also comment for benchmarks with the lowest improvements why the IPC is is hitting a wall.

In the last exercise you have to propose a cache architecture to mitigate the performance penalty of a higher memory latency. The default memory latency of SimpleScalar is 18 cycles for the initial memory chunk, and 2 cycles for each subsequent memory chunk. Each chunk has the same width as the memory bus width, i.e., each cache line transfer is transferred in multiple chunks. When we increase the initial latency to 200 cycles the performance is significantly lower. An initial latency of 200 cycles, however, is a more realistic number in current state-of-the-art processors. In this assignment we only use the *memcopy* benchmark, which has both large contiguous and more fine-grained memory access patterns. See the source file `memcpy.c` for more details.

5. (5 Points) First, simulate the IPC and calculate the AMAT before the increased memory latency and after the increased memory latency. For the AMAT calculation you can simplify by only the L2 cache for this, i.e, the AMAT in case you have an L1 miss.

   Second, propose a cache configuration that masks the performance penalty of the increased memory latency. You can increase the number of sets, set size (block/line size), associativity, of the unified L2 cache only. The default latency of the L2 cache is 6 cycles. For each doubling of the number of sets and/or associativity add 1 extra cycle of latency. Report the chosen cache configuration and resulting IPC and AMAT. Provide an explanation why you choose this cache configuration and explain how the additional memory latency is mitigated.

   For this assignment SimpleScalar has to be configured as follows: comb branch predictor, four-issue, out-of-order, default amount of resources, and a (ruu,lsq) of (32,16).

For submission, prepare one document with the results of all five exercises as the deliverable for the SimpleScalar assignment. Each exercise is worth 5 points for a total of 25 points.

4. Fachgebiet Architektur eingebetteter Systeme
Advanced Computer Architectures SS2015

Assignment start: 23.06.2015 Submission deadline: 06.07.2015

Assignment 4 - Simultaneous Multithreading 25 Points

In the final exercise you will investigate the effect of executing two threads simultaneously on the same physical core capable of Simultaneous Multithreading (SMT). SMT allows better utilization of core resources by selecting independent instruction from multiple threads simultaneously. The effectiveness, however, varies depending on the workload the threads are performing. With SMT most core resources are shared (e.g. functional units, reservation stations, reorder buffers, branch predictor entries, etc), only the architectural register files are duplicated. Because the core resources are shared, when threads content for the same resources performance of both threads reduce.

In this assignment one thread is calculating the 40th fibonacci number. Your task is to produce code for the other thread that has *as little as possible influence* on the performance of the fibonacci thread. The code sequences can use all the tricks you can think of except they must be active and running. This means no `sleep`, `printf`, or other system calls that can lead to a blocked thread. Furthermore, no inline assembly is allowed, and the fibonacci thread must not be modified.

The SMT code is located in the */afs/tu-berlin.de/units/Fak_IV/aes/aca/SMT* directory and is also available at the ISIS page. The experiments should be performed on a machine with a Intel Core-i7 3930K. This processor has the Sandy Bridge microarchitecture and the main specification can be found at `http://http://ark.intel.com/products/63697/Intel-Core-i7-3930K-Processor-12M-Cache-up-to-3_80-GHz`. More architectural information regarding the Sandy Bridge architecture can be found at `http://www.anandtech.com/show/3922/intels-sandy-bridge-architecture-exposed`. The Turbo Boost feature has been turned off for this assignment and the processor is clocked at 4GHz.

To get an account on this machine you have to give the student assistant your tubit username during the lab. The student assistant will then create you an account on this machine on which you can login with your *tubit* password. The address of this machine is *cortex.aes.tu-berlin.de* and is accessible using SSH from within the *tu-berlin* net. Also the student assistant will assign you a core number to minimize the sharing of cores during the lab. This core number should be filled as the `CPUID_SMT0` in the source file.

The deliverable is the source file of the assignment. In the file the `tf_smt1` should be documented with a brief explanation of the choices you made. The grade for this exercise will be determined by the performance of the fibonacci thread (15 points) and your explanation (10 points).