Embench

차 례

제 1 절 이 문서에 대하여

이 문서는 **IoT 장치 및 웨어러블 기기에 적합한 RISC-V 확장형 ISA 기반 경량 프로세서 기술 개발** 과제의 일부로 Spike를 이용하여 시뮬레이션이 가능하도록 Embench[1]를 수정한 내용과 수정된 벤치마크를 컴파일하고 시뮬레이션하기 위하여 작성된 스크립트에 대하여 설명한다.

[1]Embench의 원래 소스 코드는 https://www.embench.org 또는 https://github.com/embench/embench-iot에서 다운로드 받을 수 있음. 이 문서의 내용에 따라 수정된 코드는 https://github.com/semifive-rvv/main 참조.

제 2 절 Embench 사용

2.1 Prerequisites

Embench를 github repository로부터 소스 코드를 다운로드 받은 후에 컴파일한다. Embench는 python 스크립트를 이용하여 컴피일/실행되기 때문에 python (3.6 또는 이후 버전)과 pyelftools, lief가 필요하다.

Ubuntu에서 pyelftools를 설치하기 위해서는 "`sudo apt-get install python-pyelftools`"를 실행한다. lief 패키지는 "`sudo pip3 install lief`" 명령을 실행하여 설치한다. (lief 패키지는 코드 크기 벤치마킹에서 필요하다.)

2.2 스크립트

Embench를 컴파일하고, 그 결과를 Spike에서 실행하기 위해서 다음과 같은 스크립트가 제공된다. 각 스크립트의 사용법과 동작은 다음과 같다. 이 스크립트들은 EMBENCH_ROOT 디렉토리에 존재한다. 이들 스크립트 중 build_all.py, benchmark_size.py, run_all.py는 원래 Embench에서 제공되는 스크립트이며, benchmark_speed_sel.py, compile_embench.bash, run_embench.bash는 전북대학교 설계자동화연구실(SoCLab)에서 작성한 스크립트이다.

build_all.py 이 스크립트는 Embench 내의 모든 프로그램을 컴파일한다. 이 스크립트의 수행을 위해서는 "`--arch, --board, --chip`" 옵션을 반드시 모두 표시하여야 한다. (4.2 참조)

RISC-V와 Native 환경에서 필요한 옵션을 지정하여 이 스크립트를 실행할 수 있도록 compile_embench.bash를 사용할 수 있다.

benchmark_size.py 컴파일된 Embench 프로그램의 코드 크기를 출력한다. 컴파일된 실행 파일은 ELF 형식으로 이 스크립트는 실행 파일들의 크기를 정리하여 보여 준다. RISC-V와 Native 환경에서 필요한 옵션을 지정하여 이 스크립트를 실행할 수 있도록 run_embench.bash를 사용할 수 있다.(4.3 참조)

benchmark_speed.py 컴파일된 모든 Embench 프로그램을 사용자가 지정한 환경 (`--target-module` 옵션)에서 실행 한 후, 실행 결과의 오류 여부와 cycle count를 측정하여 보여준다. RISC-V 실행 파일을 Spike 환경에서 실행하기 위해서는 "`--target-module run_spike`" 옵션을 사용한다. RISC-V와 Native 환경에서 필요한 옵션을 지정하여 이 스크립트를 실행할 수 있도록 run_embench.bash를 사용할 수 있다. (4.4 참조)

benchmark_speed_sel.py 이 스크립트는 모든 벤치마크 프로그램을 실행하는 benchmark_speed.py과 달리 원하는 벤치마크 프로그램 만을 선택하여 실행할 수 있다. 벤치마크의 소스 코드는 EMBENCH_ROOT/src 아래의 각 디렉토리에 저장되어 있으며, 현재 제공되는 프로그램은 "aha-mont64, cubic, huffbench, minver, nettle-aes, nsichneu, qrduino, slre, statemate, wikisort, crc32, edn, matmult-int, nbody, nettle-sha256, picojpeg, sglib-combined, st, ud"이다.

다음 명령어는 벤치마크 중 slre, st, ud만을 실행하고 cycle count를 알려준다.

```
benchmark_speed_sel.py --target-module run_spike --absolute --bench "slre st ud"
```

주의) benchmark_size.py, benchmark_speed.py, benchmark_speed_sel.py의 기본 출력값은 baseline data로 정규화된 값이다. 만약 절대값 출력을 원하면 `--absolute` 옵션을 사용한다. 현재는 baseline data가 정확하지 않기 때문에 절대값만이 정확한 결과이다.

run_all.py 이 스크립트를 이용하면 컴파일 옵션을 변경 하면서 다양한 경우에 대하여 벤치마크 프로그램을 실행할 수 있다. 아래의 예는 RISC-V 프로세서에서 최적화 옵션을 "`-Os`, `-O0`, `-O1`, `-O2`, `-O3`"로 변경해 가면서 코드 크기와 cycle count를 측정하는 코드이다.

```
rv32_gcc_spike_runset = {
    'name' : 'RV32 GCC compiler and Spike',
    'size benchmark' : {
        'timeout' : 30,
        'arglist' : [
            './benchmark_size.py',
            '--absolute',
        ],
        'desc' : 'sized'
    },
    'speed benchmark' : {
        'timeout' : 1800,
        'arglist' : [
            './benchmark_speed.py',
            '--target-module=run_spike',
            '--absolute',
        ],
        'desc' : 'run'
    },
    'runs' : [
        { 'name' : 'rv32imc-os-save-restore',
          'arch' : 'riscv32',
          'chip' : 'generic',
          'board' : 'spike',
          'cc' : 'riscv32-unknown-elf-gcc',
          'cflags' : '-march=rv32imc -mabi=ilp32 -Os -msave-restore',
          'ldflags' : '',
        },
        { 'name' : 'rv32imc-o0',
```

```
        'arch' : 'riscv32',
        'chip' : 'generic',
        'board' : 'spike',
        'cc' : 'riscv32-unknown-elf-gcc',
        'cflags' : '-march=rv32imc -mabi=ilp32 -O0',
        'ldflags' : '',
      },
      { 'name' : 'rv32imc-o1',
        'arch' : 'riscv32',
        'chip' : 'generic',
        'board' : 'spike',
        'cc' : 'riscv32-unknown-elf-gcc',
        'cflags' : '-march=rv32imc -mabi=ilp32 -O1',
        'ldflags' : '',
      },
      { 'name' : 'rv32imc-o2',
        'arch' : 'riscv32',
        'chip' : 'generic',
        'board' : 'spike',
        'cc' : 'riscv32-unknown-elf-gcc',
        'cflags' : '-march=rv32imc -mabi=ilp32 -O2',
        'ldflags' : '',
      },
      { 'name' : 'rv32imc-o3',
        'arch' : 'riscv32',
        'chip' : 'generic',
        'board' : 'spike',
        'cc' : 'riscv32-unknown-elf-gcc',
        'cflags' : '-march=rv32imc -mabi=ilp32 -O3',
        'ldflags' : '',
      },
  ]
```

위의 코드는 다양한 벤치마크 환경에 필요한 옵션을 기술하는 스크립트 코드이다. 따라서 위 코드에서 옵션을 변경하여 원하는 환경에서 벤치마크 실행이 가능하다. 그리고 rv32_gcc_spike_runset에 기술된 옵션은 benchmark_size.py, benchmark_speed.py, benchmark_speed_sel.py를 수행하는 과정에서 옵션으로 사용된다.

- 'size benchmark'는 코드 크기 측정 벤치마크를 실행하기 위하여 필요한 옵션과 스크립트를 지정

한다.

- 'speed benchmark'는 cycle count 측정 벤치마크를 실행하기 위하여 필요한 옵션과 스크립트를 지정한다.
- 'runs'는 각 실행 모드를 중괄호 { } 사이에 지정한다. 실행 모드를 지정하기 위해서는 실행 모드의 이름 ('name'), 아키텍춰('arch'), 칩('chip'), 보드('board'), 컴파일러('cc'), 컴파일러 옵션('cflags'), 링커 옵션('ldflags') 등을 표시한다.

compile_embench.bash 이 스크립트는 옵션에 따라 4개의 모드에서 build_all.py를 호출하여 벤치마크 코드를 컴파일 한다.

- compile_embench.bash 1 : 이전 결과를 clear하고 모든 벤치마크를 RISC-V용으로 컴파일 한다.
- compile_embench.bash 2 : 이전 결과를 clear하지 않고 모든 벤치마크를 RISC-V용으로 컴파일 한다.
- compile_embench.bash 3 : 이전 결과를 clear하고 모든 벤치마크를 호스트용으로 컴파일 한다.
- compile_embench.bash 4 : 이전 결과를 clear하지 않고 모든 벤치마크를 호스트용으로 컴파일 한다.

run_embench.bash benchmark_size.py와 benchmark_speed.py를 호출하여 코드 크기와 cycle count를 측정한다.

- run_embench.bash 1 : RISC-V용으로 컴파일된 코드에 대하여 코드 크기를 측정한다.
- run_embench.bash 2 : RISC-V용으로 컴파일된 코드를 실행하여 cycle count를 측정한다.
- run_embench.bash 3 : 호스트용으로 컴파일된 코드에 대하여 코드 크기를 측정한다.
- run_embench.bash 4 : 호스트용으로 컴파일된 코드를 실행하여 cycle count를 측정한다.

2.3 실행 예

1. Prompt>compile_embench.bash 1 : 이전 결과를 clear하고, RISC-V에 대하여 전체 벤치마크를 컴파일하며, 아래와 같은 결과가 출력된다.

```
### RISCV: Clearing previous results and compiling embench...
------------------------------------------------
aha-mont64
crc32
cubic
edn
huffbench
matmult-int
minver
nbody
nettle-aes
nettle-sha256
nsichneu
picojpeg
qrduino
sglib-combined
slre
st
statemate
```

```
ud
wikisort
All benchmarks built successfully
```

2. Prompt>run_embench.bash 1 : RISC-V 실행 파일에 대하여 코드 크기 측정하며, 결과는 아래와 같이 출력된다. (코드 크기는 바이트 수로 --absolute 옵션이 사용되어, 코드 크기의 절대값이다.)

```
### RISCV-Size: Run Embench for code size on RISC-V
-----------------------------------------------
benchmark_size.py --absolute
Benchmark            size
---------            ----
aha-mont64         64,734
crc32              61,938
cubic              90,820
edn                65,212
huffbench          64,838
matmult-int        62,774
minver             67,372
nbody              68,742
nettle-aes         67,018
nettle-sha256      69,994
nsichneu           85,736
picojpeg           78,882
qrduino            74,540
sglib-combined     76,722
slre               67,916
st                 68,738
statemate          66,666
ud                 65,692
wikisort           81,058
---------               -----
Geometric mean             70,615
Geometric SD                 1.11
Geometric range          14903.50
All benchmarks sized successfully
```

3. Prompt>run_embench.bash 2 : RISC-V 실행 파일을 Spike에서 실행 한 후 cycle count 측정하고 아래와 같이 출력한다. 아래의 출력은 cycle count의 절대값 (baseline machine의 값에 대한 정규화된 값이 아님)을 표시한다.

```
### RISCV-Speed: Run Embench for cycle count on RISC-V ...
------------------------------------------------
benchmark_speed.py --target-module run_spike --absolute
Benchmark               Speed
---------               -----
aha-mont64             13,680,296
crc32                   6,621,544
cubic                  15,900,851
edn                    12,500,636
huffbench               7,190,345
matmult-int            18,466,097
minver                 68,991,536
nbody                   7,892,260
nettle-aes              7,509,023
nettle-sha256           6,255,791
nsichneu                3,855,531
picojpeg               11,121,887
```

```
qrduino                  6,490,216
sglib-combined           6,348,054
slre                     6,074,461
st                      12,707,138
statemate                1,706,757
ud                      12,719,709
wikisort                 3,246,168
---------                -----
Geometric mean           8,614,251
Geometric SD                  2.12
Geometric range        14217890.69
All benchmarks run successfully
```

4. `Prompt>benchmark_speed_sel.py --target-module run_spike --absolute --bench "slre st ud"` : slre, st, ud의 RISC-V 실행 파일을 Spike에서 실행 한 후 cycle count 측정하여 출력한다.

```
Benchmark                Speed
---------                -----
slre                     6,074,461
st                      12,707,138
ud                      12,719,709
---------                -----
Geometric mean           9,939,035
Geometric SD                  1.42
Geometric range         7061481.52
All benchmarks run successfully
```

5. `Prompt>run_all.py --rv32-gcc-spike` : -Os, -O0, -O1, -O2, -O3에 대하여 벤치마크 수행하고 실행 결과는 EMBENCH_ROOT/results에 저장된다.

```
RV32 GCC compiler and Spike
  rv32imc-os-save-restore
  rv32imc-o0
  rv32imc-o1
  rv32imc-o2
  rv32imc-o3
```

제 3 절 Embench 디렉토리 구조

The top level directory (**EMBENCH_ROOT**) contains Python scripts to build and execute the benchmarks. The following are the key top level directories.

- config: containing a directory for each architecture supported, and within that directory subdirectories for board and cpu descriptions. Configuation data can be provided for individual CPUs and individual boards.
- src: The source for the benchmarks, one directory per benchmark.
- support: The generic wrapper code for benchmarks, including substitutes for some library and emulation functions.

- support: The generic wrapper code for benchmarks, including substitutes for some library and emulation functions.
- pylib: Support code for the python scripts
- results: Results generated by `run_all.py` are stored.

주의: 본 문서는 Ubuntu linux가 운영체제인 것을 가정하여 기술됨.

제 4 절 Building and Running Embench

4.1 Configuration

Embench는 다음 방법을 사용하여 configuration이 가능하다. 벤치마크를 configuration하기 위하여 아래의 방법 중 여러 가지 방법을 동시에 사용 가능하며, 이 경우에는 나중에 기술된 configuration이 우선 순위를 가진다. command line에 표시된 flag의 경우에는 나중에 기술된 flag의 우선 순위가 높다.

- default values in the build script
- an architecture specific configuration
  - found in config/<architecture>/arch.cfg
  - for example config/riscv32/arch.cfg
- a chip specific configuration file
  - found in config/<architecture>/chips/<chip>/chip.cfg
  - for example config/riscv32/chips/speed-test/chip.cfg;
- a board specific configuration file
  - found in config/<architecture>/boards/<board>/board.cfg
  - for example config/riscv32/boards/ri5cyverilator/board.cfg;
- on the command line to the build script

위의 configuration 파일 외에 칩과 보드를 기술한 헤더 파일과 소스 파일(보드/칩 특화 파일)이 기술되어 사용될 수도 있다. (즉 보드나 칩의 기능이 변경된 경우에는 이 파일들을 변경 내용에 맞게 수정하여야 한다.)

- a chip specific header in config/<architecture>/chips/<chip>/chipsupport.h
  - for example config/riscv32/chips/speed-test/chipsupport.h;
- a cchip specific code in config/<architecture>/chips/<chip>/chipsupport.c
  - for example config/riscv32/chips/speed-test/chipsupport.c;

- a board specific header in config/<architecture>/boards/<board>/boardsupport.h
  - for example config/riscv32/boards/ri5cyverilator/boardsupport.h;
- a board specific code in config/<architecture>/boards/<board>/boardsupport.c
  - for example config/riscv32/boards/ri5cyverilator/boardsupport.c

4.1.1 Configuration Parameters

Configuration 파일 (architecture, board, chip)에서는 다음의 파라미터들이 python의 변수로 정의되고, 이 변수들에 대하여 원하는 값을 할당(assign)함으로써 벤치마크를 configuration한다. 아래 변수들이 모두 값이 지정되어야 할 필요는 없다. 따라서 아래 변수들의 값을 변경할 필요가 없는 경우(즉 모두 default 값이 사용되는 경우)에는 configuration 파일이 빈 파일이 될 수도 있다.

cc The C compiler to use. Default value cc.

ld The linker to use. Default value is be the same value as was set for cc, which means with most modern compilers there is no need to set ld, since the compiler can act as a linker driver.

cflags A Python list of additional compiler flags, which are appended to any other compiler flags. Default is the empty list.

ldflags A Python list of additional linker flags, which are appended to any other linker flags. Default is the empty list.

cc-define1-pattern A Python formatted string pattern with positional arguments to be used when defining a constant on the compiler command line. Default value is -D0.

cc-define2-pattern A Python formatted string pattern with positional arguments to be used when defining a constant to a specific value on the compiler command line. Default value is -D0=1.

cc-incdir-pattern A Python formatted string pattern with positional arguments to be used when specifying an include directory on the compiler command line. Default value -I0.

cc-input-pattern A Python formatted string pattern with positional arguments to be used when specifying the input file on the compiler command line. Default value 0.

cc-output-pattern A Python formatted string pattern with positional arguments to be used when specifying the output file on the compiler command line. Default value -o 0.

ld-input-pattern A Python formatted string pattern with positional arguments to be used when specifying the input file on the linker command line. Default value 0.

ld-output-pattern A Python formatted string pattern with positional arguments to be used when specifying the output file on the linker command line. Default value -o 0.

user-libs A list of libraries to be appended to the linker command line. The libraries may be absolute file names or arguments to the linker. In the latter case corresponding arguments in ldflags may be needed. For example with GCC or Clang/LLVM if -l flags are used in user_libs, then -L flags may be needed in ldflags. Default value is the empty list.

dummy-libs A list of dummy libraries to be used (for example if system libraries have been disabled through options in ldflags). Dummy libraries have their source in the support subdirectory. Thus if crt0 is specified, there should be a source file dummy-crt0.c in the support directory. Default value is the empty list.

cpu-mhz The clock rate of the target in MHz. Default value 1.

warmup-heat How many times the benchmark code should be run to warm up the caches. Default value 1.

timeout The maximum time (in seconds) allowed for the compiler or the linker to run for each invocation. Default value 5.

보드/칩 특화 파일은 벤치마킹하려는 보드와 칩의 기능을 기술하는 핵심적인 코드가 포함된다. 대개의 경우 칩 특화 파일은 특별한 내용을 포함하지 않는다. 보드 특화 헤더 파일 (boardsupport.h)에서는 다음과 같이 보드의 클록 주파수를 설정하는 코드를 포함하고 있다.

```
#define CPU_MHZ 1
```

보드 특화 C 파일 (boardsupport.c)은 다음의 3가지 함수를 정의하고 있다.

- void initialise_board (): Called to set up the board;
- void start_trigger (): Called when we start timing the benchmark, to start any board specific timing mechanism.
- void stop_trigger (): Called when we stop timing the benchmark, to stop any board specific timing mechanism.

4.1.2 Configuration for RISCV32 and Spike

Embench의 원 소스 코드에서는 호스트와 타겟이 다른 경우에 대하여, 벤치마크 프로그램을 실행하기 위한 방법으로 시뮬레이션 환경으로 gdbserver와 Verilator를 사용하는 방법과 STM32F4 보드를 사용하는 방법 만을 지원하고 있다. 따라서 Spike를 이용하여 벤치마크 코드를 수행하기 위해서는 다음과 같은 소스 코드의 변경이 필요하다.

Boardsupport 기술 boardsupport.c과 board.cfg에서 시뮬레이션에 해당하는 보드에 대한 기술로 이 경우에는 Spike에서 cycle count에서 측정하는 코드를 기술해야 한다.

- cycle count를 기술하는 변수에 대한 extern 선언 (boardsupport.c)

```
extern unsigned long spike_start_cycle, spike_stop_cycle;
```

- 벤치마크 코드 실행 직전의 cycle count를 spike_start_cycle에 기록 (boardsupport.c)

```
void __attribute__ ((noinline)) __attribute__ ((externally_visible))
start_trigger() {
   __asm__ volatile ("rdcycle %0" : "=r" (spike_start_cycle));
}
```

- 벤치마크 코드 실행 직후의 cycle count를 spike_stop_cycle에 기록 (boardsupport.c)

```
void __attribute__ ((noinline)) __attribute__ ((externally_visible))
stop_trigger () {
  __asm__ volatile ("rdcycle %0" : "=r" (spike_stop_cycle));
}
```

- main 함수에서 cycle count를 STDOUT에 출력하는 코드가 사용되도록 매크로 정의 (board.cfg)

```
cflags = [
      '-DSPIKE_SIM=1'
]
```

EMBENCH_ROOT/support/main.c 수정 각 벤치마크의 소스 코드는 EMBENCH_ROOT/src의 각 디렉토리에 저장되어 있으며, 이 소스 코드는 EMBENCH_ROOT/support 아래의 support 코드와 같이 컴파일 되어 실행파일이 생성된다. 이 support 코드 파일은 "beebsc.c beebsc.h board.c chip.c dummy-crt0.c dummy-libc.c dummy-libgcc.c dummy-libm.c main.c support.h"이다. 이 중에서 main.c 파일은 main 함수를 포함하고 있으며, 다음과 같이 수정한다.

- cycle count를 기록하는 변수 선언 한다.

  ```
  unsigned long spike_start_cycle, spike_stop_cycle;
  ```

  spike_start_cycle, spike_stop_cycle는 각각 벤치마크 코드의 수행을 시작하는 시점의 cycle count와 수행이 끝나는 시점의 cycle count를 저장하고 있으며, 경과된 cycle count의 값은 spike_stop_cycle - spike_start_cycle로 계산된다. main 함수에서는 다음과 같이 벤치마크 코드 실행의 직전과 직후에 cycle count를 기록하는 함수인 start_trigger, stop_trigger를 호출하며, 이 함수 안에서 spike_start_cycle, spike_stop_cycle에 cycle count가 저장된다.

  ```
  start_trigger ();
  result = benchmark ();
  stop_trigger ();
  ```

- cycle count를 Spike 시뮬레이터의 STDOUT으로 출력하는 코드 추가:

```
#ifdef SPIKE_SIM
    printf("CYCLES=%ld\n", spike_stop_cycle - spike_start_cycle);
#endif
```

4.2 Building the benchmarks

Embench는 `build_all.py` 스크립트를 실행하여 컴파일 가능하며, 이 스크립트는 다음과 같은 옵션이 적용 가능하다.

–***arch*** *This mandatory argument specifies the architecture for which the benchmarks are to be built. It corresponds to a directory name in the main config directory.*

–***chip*** *This mandatory argument specifies the chip being used and corresponds to a directory within the chips subdirectory of the architecture configuration directory.*

–***board*** *This mandatory argument specifies the board being used and corresponds to a directory within the boards subdirectory of the architecture configuration directory.*

–**builddir** The programs are build out of tree, this specifies the directory in which to build. It may be an absolute or relative directory name; if the latter, it will be relative to the top level directory of the repository. Default value bd.

–**logdir** A log file is created with detailed information about the build. This specifies the directory in which to place the log file. It may be an absolute or relative directory name; if the latter, it will be relative to the top level directory of the repository. Default value logs.

–**cc** The C compiler to be used. Default value cc.

–**ld** The linker to be used. Default value the same value as for –cc

–**cflags** A space separated list of additional C flags to be appended to the compiler flags. Default value empty.

–**ldflags** A space separated list of additional linker flags to be appended to the linker flags. Default value empty.

–**cc-define1-pattern** A Python formatted string pattern with positional arguments to be used when defining a constant on the compiler command line. Default value -D0.

–**cc-define2-pattern** A Python formatted string pattern with positional arguments to be used when defining a constant to a specific value on the compiler command line. Default value -D0=1.

–**cc-incdir-pattern** A Python formatted string pattern with positional arguments to be used when specifying an include directory on the compiler command line. Default value -I0.

–**cc-input-pattern** A Python formatted string pattern with positional arguments to be used when specifying the input file on the compiler command line. Default value 0.

–**cc-output-pattern** A Python formatted string pattern with positional arguments to be used when specifying the output file on the compiler command line. Default value -o 0.

–**ld-input-pattern** A Python formatted string pattern with positional arguments to be used when specifying the input file on the linker command line. Default value 0.

–**ld-output-pattern** A Python formatted string pattern with positional arguments to be used when specifying the output file on the linker command line. Default value -o 0.

–**user-libs** A space separated list of libraries to be appended to the linker command line. The libraries may be absolute file names or arguments to the linker. In the latter case corresponding arguments in –ldflags may be needed. For example with GCC or Clang/LLVM if -l flags are used in –user_libs, then -L flags may be needed in –ldflags. Default value empty.

–**dummy-libs** A space separated list of dummy libraries to be used (for example if system libraries have been disabled through options in –ldflags). Dummy libraries have their source in the support subdirectory. Thus if crt0 is specified, there should be a source file dummy-crt0.c in the support directory. Default value empty.

–**cpu-mhz** The clock rate of the target in MHz. Default value 1.

–**warmup-heat** How many times the benchmark code should be run to warm up the caches. Default value 1.

–**timeout** The maximum time (in seconds) allowed for the compiler or the linker to run for each invocation. Default value 5.

–**clean** Delete all intermediaries and final files from any previous runs of the script.

–**help** Provide help on the arguments.

4.2.1 Command for Compiling Embench

여기서는 타겟머신이 native와 riscv32 경우에 Embench을 컴파일하는 명령을 기술한다.

native 이 경우는 호스트와 타겟이 같은 경우로 벤치마크 코드를 컴파일하여 호스트에서 실행 가능한 실행파일을 생성하는 경우이다. 따라서 호스트 머신의 컴파일러 (대개의 경우 GCC 혹은 LLVM)를 사용하여 호스트에서 실행 가능한 실행파일을 생성한다.

`build_all.py` 스크립트는 다음과 같은 옵션을 사용하여 실행한다.

```
build_all.py --arch native --board default --chip default --clean
```

`--clean` 옵션을 사용하면 이전의 결과를 모두 지우고 다시 컴파일하게 된다.

수행 속도에 대한 벤치마킹을 위하여 호스트와 타겟이 같기 때문에 생성된 실행 파일은 시스템에서 직접 수행된다.

riscv32 호스트는 CPU는 X86 프로세서와 운영체제는 Ubuntu이며, 타겟머신은 RISC-V 32비트이다. 컴파일로 생성되는 실행 파일은 RISC-V 프로세서에서 실행되는 실행파일이며, 추후 수행 속도 벤치마킹을 위하여 적당한 시뮬레이터 (`riscv32-unknown-elf-run` 또는 `spike`)를 지정하여야 한다.

`build_all.py` 스크립트는 다음과 같은 옵션을 사용하여 실행한다.

```
build_all.py --arch riscv32 --chip generic --board spike
               --cc /home/jong/Projects/RISCV/bin/riscv32-unknown-elf-gcc --clean
```

`--clean` 옵션을 사용하면 이전의 결과를 모두 지우고 다시 컴파일하게 된다.

4.3 Running the benchmark for code size

Embench를 이용하여 코드 크기에 대한 벤치마킹을 진행하기 위해서는 benchmark_size.py 스크립트를 사용한다. benchmark_size.py에 적용 가능한 옵션은 다음과 같다.

–**format** Specifies the file format of executable executable and/or object files. The option are elf and macho. Default value elf.

–**builddir** The programs are build out of tree, this specifies the directory in which the programs were built. It may be an absolute or relative directory name; if the latter, it will be relative to the top level directory of the repository. Default value bd.

–**logdir** A log file is created with detailed information about the benchmark run. This specifies the directory in which to place the log file. It may be an absolute or relative directory name; if the latter, it will be relative to the top level directory of the repository. Default value logs.

–**baselinedir <dir>** Specifies the directory in which reference size data can be found. May be an absolute or relative directory name. If it is relative then it will be interpreted as relative to the top level directory of the repository. The default value is baseline-data, and size data will be sourced from baseline-data/size.json.

–**relative or –absolute** If –relative is specified, present benchmark results relative to the baseline architecture. If –absolute is specified, present absolute benchmark results. If neither is specified, present relative results, because this is the defined norm for Embench.

–**text** A space separated list of sections containing code. Default value for elf format files .text; for macho format files __text.

–**data** A space separated list of sections containing non-zero initialized writable data. Default value for elf format files .data; for macho format files __data. The option –metric with the respective section type needs to be used in order to have the sections added to the size calculation.

–**rodata** A space separated list of sections containing read only data. Default value for elf format files .rodata; for macho format files __cstring __const. The option –metric with the respective section type needs to be used in order to have the sections added to the size calculation.

–**bss** A space separated list of sections containing zero initialized data. Default value for elf format files .bss, for macho format files __bss. The option –metric with the respective section type needs to be used in order to have the sections added to the size calculation.

–**metric** A space separated list of section types to include when calculating the benchmark metric. Any section listed with the options –text, –data, –rodata and –bss is included in the respective section type. Permitted values ares text, data, rodata, bss. Default value text.

–**text-output** Output the text in a plain text format. This is the default.

–**json-output** Output the results in json format, instead of the default plain text format.

–**baseline-output** Output results in a format suitable for use as baseline data instead of the default text format. This can be used instead of the reference data in baseline-data/size.json.

–**help** Provide help on the arguments.

4.4 Running the benchmark for code speed

수행 속도에 대한 벤치마킹을 실행하는 python 스크립트는 benchmark_speed.py이며, 다음의 옵션이 적용 가능하다.

–***target-module <target module>*** This mandatory argument specifies a python module in the pylib directory with definitions of routines to run the benchmark. Note that the argument specifies the name of module (e.g. run_stm32f4-discovery) not the name of the file that contains the module (e.g. run_stm32f4-discovery.py).

–**builddir** The programs are build out of tree, this specifies the directory in which the programs were built. It may be an absolute or relative directory name; if the latter, it will be relative to the top level directory of the repository. Default value bd.

–**logdir** A log file is created with detailed information about the build. This specifies the directory in which to place the log file. It may be an absolute or relative directory name; if the latter, it will be relative to the top level directory of the repository. Default value logs.

–**baselinedir <dir>** Specifies the directory in which reference speed data can be found. May be an absolute or relative directory name. If it is relative then it will be interpreted as relative to the top level directory of the repository. The default value is baseline-data, and speed data will be sourced from baseline-data/speed.json.

–**relative or –absolute** If –relative is specified, present benchmark results relative to the baseline architecture. If –absolute is specified, present absolute benchmark results. If neither is specified, present relative results, because this is the defined norm for Embench.

–**text-output** Output the text in a plain text format. This is the default.

–**json-output** Output the results in json format, instead of the default plain text format.

–**baseline-output** Output results in a format suitable for use as baseline data instead of the default text format. This can be used instead of the reference data in baseline-data/speed.json.

–**timeout** The maximum time (in seconds) allowed for each benchmark program to run. Default value 30.

–**help** Provide help on the arguments.

위 옵션 중에서 –*target-module*를 이용하여 벤치마크 코드의 실행 환경을 지정한다. 이 옵션의 값은 target machine (시뮬레이터, 하드웨어 보드 등)에서 벤치마크 코드를 실행하는 방법이 기술된 python 모듈을 지정한다. 현재 pylib 디렉토리에는 4개의 모듈이 작성되어 있다. ("run_gdbserver_sim.py, run_mac.py, run_native.py, run_stm32f4-discovery.py") run_gdbserver_sim.py에서는 GDB와 시뮬레이터를 이용하여 벤치마크 코드를 수행하기 위한 명령어와 옵션을 기술하고 있으며, run_mac.py에는 Mac OS X에서 벤치마크 코드를 수행하는 명령어와 옵션을 기술하고 있다. run_native.py에는 호스트에서 코드를 수행하기 위한 명령을 포함하고 있으며, run_stm32f4-discovery.py는 stm32f4-discovery 보드에서 벤치마크 코드를 실행하기 위한 명령을 포함하고 있다.

4.1.2에 기술된대로 RISC-V와 Spike를 이용하여 벤치마크 코드를 실행하기 위하여, run_spike.py가 추가 되었다. run_spike.py에서는 컴파일된 벤치마크 코드를 Spike 상에서 실행하도록 명령을 생성하고 수행한다. 또한 벤치마크 코드 결과 STDOUT으로 출력되는 cycle count를 추출하여 각 벤치마크 코드를 실행하는데 소요된 cycle count를 정리하여 보여준다.