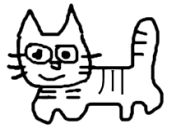
Xbyak_aarch64; Just-In-Time Assembler for ARMv8-A and Scalable Vector Extension

Kentaro Kawakami <kawakami.k@fujitsu.com>
Fujitsu Laboratories Ltd., Kawasaki, Japan

Linaro Connect
VIRTUAL • SPRING 2021

About Me



My GitHub account icon

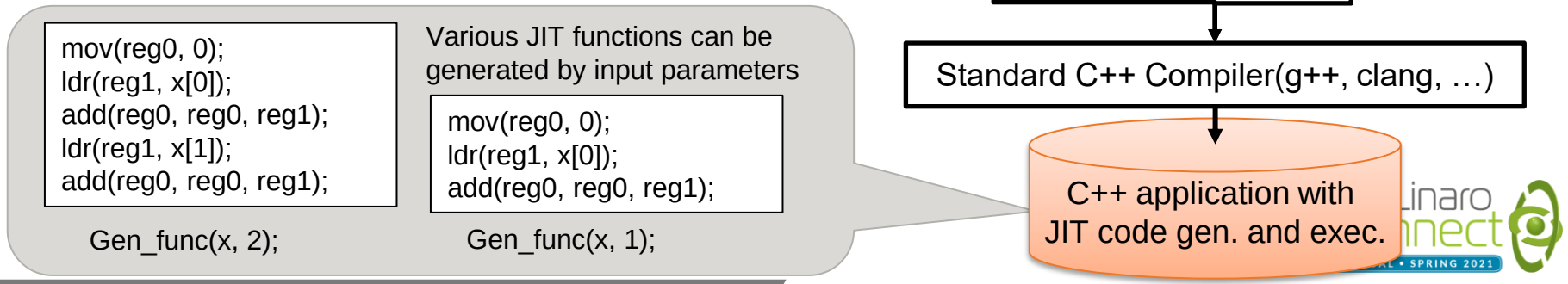
- Kentaro Kawakami <kawakami.k@fujitsu.com>
  - GitHub account: kawakami-k
  - Senior Researcher at Platform Innovation project, Fujitsu Laboratories Ltd., Japan
  - Engaged in R&D of AI software for Arm high-performance computing
  - Developing the deep learning software stack for Fugaku, the world first Arm ISA-based supercomputer, for the last two years

Table of Contents

What is Xbyak_aarch64?

What is Xbyak_aarch64?

- A JIT assembler for AArch64
- Enables to assemble AArch64 mnemonic at runtime
- We can make generators that produce instruction sequence
- Just-In-Time (JIT) functions are generated by the generators at runtime
- Based on Xbyak that is for x64 CPUs by S. Mitsunari (Cybozu Labs. Inc.)



Minimal Sample Code

Red bold texts are the functions, classes and instances provided by Xbyak_aarch64.

```
/* Write source code in C++11 or later */
#include <xbyak_aarch64/xbyak_aarch64.h>
using namespace Xbyak_aarch64;
class Generator : public CodeGenerator {
public:
  Generator() {
    add(w0, w1, w0);
    ret();
  }
};
int main() {
  Generator gen;
  gen.ready();
  auto f = gen.getCode<int (*)(int, int)>();
  int a = 3, b = 4;
  printf("%d + %d = %d¥n", a, b, f(a, b));
  return 0;
}
```

Include “Xbyak_aarch64.h”.

Define your class inheriting “CodeGenerator”.

Implement instruction sequence you want to do.

Machine code sequence, “add” and “ret” is generated.

Define a function pointer for the machine code sequence.

The machine code sequence can be called as a C++ function.

make & execute

```
> ./a.out
> 3 + 4 = 7
>
```

Machine Code Sequence Generation

```
Generator() {
  add(w0, w1, w0);
  ret();
}
```

Write machine code sequences **at runtime**

Red bold texts are the functions, classes and instances provided by Xbyak_aarch64.

We call these functions as **the mnemonic functions.**

- The function name is one of the mnemonic of ARMv8-A + SVE instruction set.
- Each function outputs a single machine code correspond to the function name, such as “add”, “sub”, “ldr”, “str”, “ret”, etc.
- The instruction operands can be indicated by the function arguments.

Memory

```
Address,          Machine code
0xXXXXXXXX0000,   0x0b000020
0xXXXXXXXX0004,   0xd65f03c0
```

=

```
Disassembled code
add w0, w1, w0
ret
```

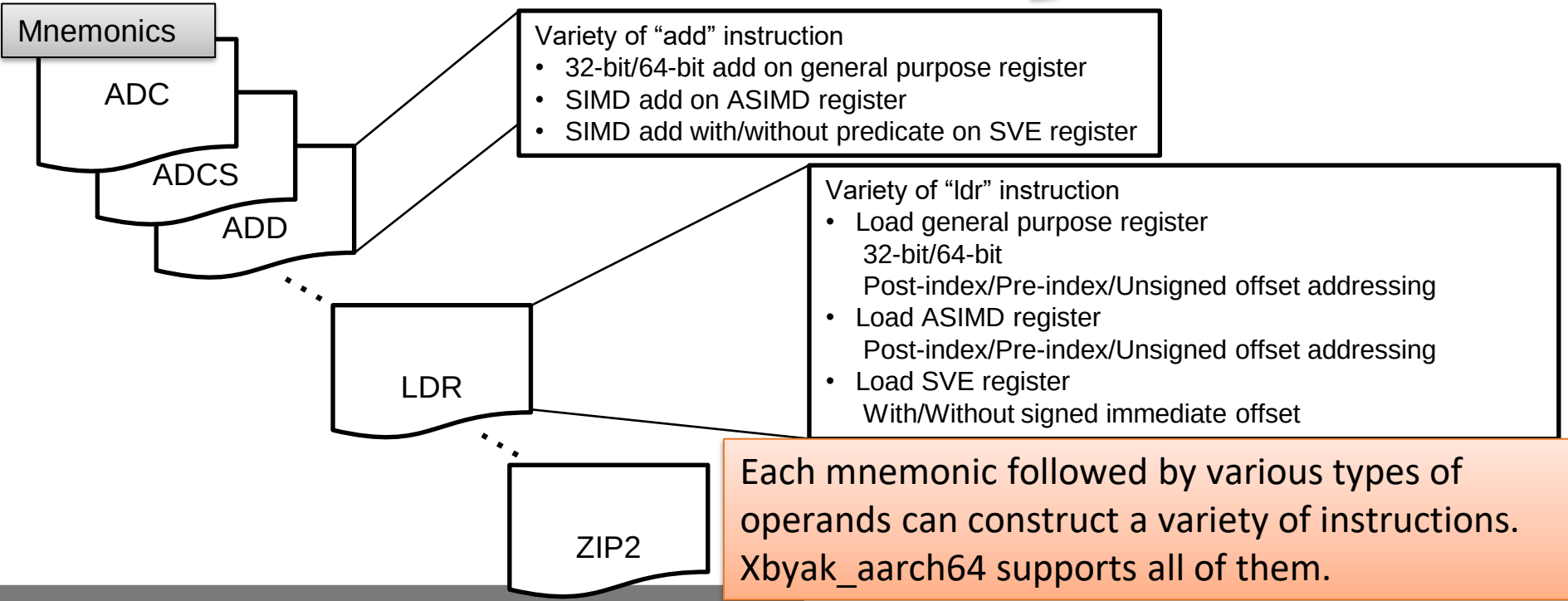
w0, w1: 32-bit general purpose registers

The machine code sequence can be called as a function.

Supported Instructions

- ARMv8-A, ARMv8.1, ARMv8.2, ARMv8.3 instructions
- Scalable Vector Extension (SVE) instructions

~1K mnemonics



Each mnemonic followed by various types of operands can construct a variety of instructions. Xbyak_aarch64 supports all of them.

Advantage of Xbyak_aarch64 compared to Existing Assembler

- Easier to write assembly code
  - Simple assembly description in C++ syntax
  - Loop unrolling is easy to describe
- Optimization using runtime parameters
  - It is possible to change the instructions using parameters

Dynamical unrolling

```
for (int j = 0; j < 15; ++j)
 fmla(ZRegS(j), PReg(0), ZRegS(j + 15), ZRegS(31));
```

Implementation becomes simpler.
(The above code generates 15 “fmla” instructions.)

```
fmla z0.s, p0.s, z15.s, z31.s
fmla z1.s, p0.s, z16.s, z31.s
….
fmla z14.s, p0.s, z29.s, z31.s
```

} x15

Dynamical instruction selection

```
if ( isPowOfTwo(param) ) {
  int bitPos = 0;
  do { bitPos +=1;
  } while  (param = (param >> 1));
  lsr(r0, r1, bitPos);  // Logical shift right
} else {
  udiv(r0, r1, param); // unsigned division
}
```

Considering execution latency,
“lsl” is more preferable than “udiv”.

Performance Comparison

- Environment: FX700 / GCC 8.3.1
  - CPU: A64FX (ARMv8-A + 512-bit SVE)
  - Compile options: -march=armv8.2-a+sve -fopenmp -O3
- Measurement conditions
  - Reduction operation for *N*-size array (*N* = 512)
  - iterated 10 million times

Reference code

```
float reduction(float* A){
  s = 0;
  for(i = 0; i < N; i++)
    s += A[i];
  return s;
}
```

Reference code + pragma

```
float reduction(float* A){
  s = 0;
  #pragma omp simd reduction (+:s)
  for(i = 0; i < N; i++)
    s += A[i];
  return s;
}
```

JIT implementation with Xbyak_aarch64

```
void generate(int N) {
  size_t offset = sizeof(float) * 16;
  int numZregs = N/16;
  ptrue(PRegS(0));
  for(int i = 0; i < numZregs; i++){
    add (x1, x0, i * offset);
    ldr(ZReg(i), ptr(x1));
  }
  for(int i = numZregs; i > 0; i=i>>1){
    for(int j =0; j < (i/2); j++){
      if((j+(i/2)) < numZregs)
        fadd(ZRegS(j), ZRegS(j), ZRegS(j+(i/2)));
  }}
  faddv(SReg(0), PReg(0), ZRegS(0));
  ret();
}
```

Execution time [sec]

23.24 → 6.30 (x51.6 times faster) → 0.45 (x14 times faster)

Proven working configuration

Main Development Target

(Since Xbyak_aarch64 is an OSS, it is basically provided as is.)

- H/W: Fugaku, Fujitsu PRIMEHPC FX1000/FX700
  - CPU: Fujitsu A64FX, designed for high-performance computing and complies with the ARMv8-A architecture profile and the Scalable Vector Extension (SVE)
- Compiler: FCC(Fujitsu C/C++ compiler)/GCC/LLVM
- Language: C++11 or later
- OS: Linux (RedHat Enterprise Linux 8.x)

Proven working configurations

(Since Xbyak_aarch64 is an OSS, it is basically provided as is.)

H/W	CPU	OS (64-bit)	Compiler	Note
FX1000 *1	Fujitsu A64FX	RedHat Enterprise Linux 8.x	FCC*2	Well-tested oneDNN *3 works
FX700 *1	Fujitsu A64FX	CentOS 8	FCC/GCC/LLVM	
IA server	QEMU 5.0.0 (Linux user mode)	(Host OS running on IA server) Ubuntu 16.04.6 LTS	GCC	oneDNN works
MAC mini, 2020	Apple M1	macOS Big Sur	Apple clang	oneDNN works
Raspberry Pi3 Model B Rev 1.2	Broadcom BCM2835	Ubuntu 18.04.4 LTS	GCC	Some samples of Xbyak_aarch64 works *4

*1 https://www.fujitsu.com/jp/products/computing/servers/supercomputer/ (in Japanese)
*2 C/C++ compiler of FUJITSU Software Compiler Package
*3 oneDNN is one of the applications that uses up all the functionality of Xbyak/Xbyak_aarch64.
*4 A limited number of sample programs has been tested.

Usage of Xbyak_aarch64

Usage of Mnemonic Functions

- Prototype declaration of the mnemonic functions
  - https://github.com/fujitsu/xbyak_aarch64/blob/main/xbyak_aarch64/xbyak_aarch64_mnemonic_def.h

```
void add(const WReg &rd, const WReg &rn, const uint32_t imm, const uint32_t sh = 0);
void add(const WReg &rd, const WReg &rn, const WReg &rm, const ShMod shmod = NONE, const uint32_t sh = 0);
void sub(const XReg &rd, const XReg &rn, const uint32_t imm, const uint32_t sh = 0);
void sub(const XReg &rd, const XReg &rn, const XReg &rm, const ShMod shmod = NONE, const uint32_t sh = 0);
void ldr(const WReg &rt, const AdrReg &adr);
void ldr(const XReg &rt, const AdrReg &adr);
void fmul(const ZRegH &zdn, const _PReg &pg, const ZRegH &zm);
```

- Usage samples
  - https://github.com/fujitsu/xbyak_aarch64/tree/main/sample/mnemonic_syntax/nm.make*.cpp

```
add(w0, w0, 0x2aa); dump();
add(w0, w0, w2); dump();
sub(x0, x0, 0x2aa); dump();
sub(x0, x0, x2); dump();
ldr(w0, ptr( x3 ) ); dump();
ldr(x0, ptr( x3 ) ); dump();
fmul(z0.h, p7/T_m, z7.h); dump();
```

General Purpose Register Class

Class name defined in Xbyak_aarch64	Pre-instantiated variable	Remarks
WReg	w0, w1, ..., w30	32-bit general purpose registers
	wsp, wzr	32-bit stack pointer, zero register
XReg	x0, x1, ..., x30	64-bit general purpose registers
	sp, xzr	64-bit stack pointer, zero register

```
WReg dstReg(0);
WReg srcReg0(1);
WReg srcReg1(2);
add(dstReg, srcReg0, srcReg1);
add(w0, w1, w2);
for(size_t i=0; i<16; i++)
  add(WReg(i), WReg(i), WReg(i+1));
```

(A) Register instances can be freely defined.

(B) These two line generate the same machine code of “add w0, w1, w2”.

(C) Register can be instantiated on the fly.

Xbyak_aarch64 also defines the classes and has the pre-instantiated variables for V (128-bit SIMD), Z (SVE), P (scalable predicate) registers.
Please refer README.md of Xbyak_aarch64.

Passing parameters to JIT-ed code/ Receiving return value from JIT-ed code

- As JIT-ed code complies the procedure call standard of AArch64, JIT-ed code can freely exchange parameters with the code generated by compiler.

Table 2, General purpose registers and AAPCS64 usage

Register	Special	Role in the procedure call standard
SP		The Stack Pointer.
r30	LR	The Link Register.
r29	FP	The Frame Pointer
r19…r28		Callee-saved registers
r18		The Platform Register, if needed; otherwise a temporary register. See notes.
r17	IP1	The second intra-procedure-call temporary register (can be used by call veneers and PLT code); at other times may be used as a temporary register.
r16	IP0	The first intra-procedure-call scratch register (can be used by call veneers and PLT code); at other times may be used as a temporary register.
r9…r15		Temporary registers
r8		Indirect result location register
r0…r7		Parameter/result registers

From “Procedure Call Standard for the Arm 64-bit Architecture (AArch64)”

```
Generator() {
  add(w0, w1, w0);
  ret();
}
```

- The first and second parameters are passed by r0(w0), r1(w1).
- The return value is passed by r0(w0).

Register usage in JIT-ed code

Table 2, General purpose registers and AAPCS64 usage

Register	Special	Role in the procedure call standard
SP		The Stack Pointer.
r30	LR	The Link Register.
r29	FP	The Frame Pointer
r19…r28		Callee-saved registers
r18		The Platform Register, if needed; otherwise a temporary register. See notes.
r17	IP1	The second intra-procedure-call temporary register (can be used by call veneers and PLT code); at other times may be used as a temporary register.
r16	IP0	The first intra-procedure-call scratch register (can be used by call veneers and PLT code); at other times may be used as a temporary register.
r9…r15		Temporary registers
r8		Indirect result location register
r0…r7		Parameter/result registers

JIT-ed code
- must save the values on these registers to the stack before use them,
- must restore them before “ret” instruction.

JIT-ed code can be freely use these registers.

“Procedure Call Standard” also defines the usage for V (128-bit SIMD) registers, Z (SVE) resisters and P (scalable predicate) registers of SVE, please refer them.

Register usage in JIT-ed code

```
Generator() {
  /* stp: store register pair
     pre_ptr: Pre-index addressing
     sp: stack pointer register */
  for(size_t i=19; i<=28; i+=2)
    stp(XReg(i), XReg(i+1), pre_ptr(sp, -16));

  /* Implement what you want to do
     with x0 - x7, x9 - x15, x19 - x28. */

  /* ldp: load register pair
     post_ptr: Post-index addressing */
  for(size_t i=28; i>=19; i-=2)
    ldp(XReg(i-1), XReg(i), post_ptr(sp, 16));

  ret();
}
```

Red bold texts are the functions, classes and instances provided by Xbyak_aarch64.

Save the registers before use them

Restore the registers after use them

Label and Branch Instructions

Red bold texts are the functions, classes and instances provided by Xbyak_aarch64.

```
Generator() {
  Label L1, L2;    // Instancing Label class of Xbyak_aarch64.
  L(L1);           // L function of Xbyak_aarch64 registers JIT-ed code address of this position to Label L1.
  add(w0, w1, w0);
  cmp(w0, 13);     // Compare the register w0 value to the immediate value 13.
  b(EQ, L2);       // Branch to L2, if the register w0 value == 13.
  sub(w1, w1, 1); // Decrement loop counter value.
  b(L1);           // Unconditional branch.
  L(L2);
  ret();
}
```

JIT-ed code

```
B+> 0xffffbe7a0000  add     w0, w1, w0
    0xffffbe7a0004  cmp     w0, #0xd
    0xffffbe7a0008  b.eq    0xffffbe7a0014  // b.none
    0xffffbe7a000c  sub     w1, w1, #0x1
    0xffffbe7a0010  b       0xffffbe7a0000
    0xffffbe7a0014  ret
```

Referencing Static Table

```
#include <xbyak_aarch64/xbyak_aarch64.h>
using namespace Xbyak_aarch64;
class Generator : public CodeGenerator {
public:
  Generator() {
    mov(x1, reinterpret_cast<uint64_t>(table));
    add(x1, x1, x0, LSL, 2);
    ldr(w0, ptr(x1));
    ret();
  }

private:
  const uint32_t table[4] =
    {0x0, 0x11111111, 0x22222222, 0x33333333};
};
int main() {

  Generator gen;
  gen.ready();
  auto f = gen.getCode<uint32_t (*)(uint32_t)>();
  uint32_t a = 2;
  printf("table[%d] = 0x%08x\n", a, f(a));
}
```

JIT-ed code

```
B+  0xffffbe7a0000  mov    x1, #0xf110                    // #61712
    0xffffbe7a0004  movk   x1, #0xffff, lsl #16
    0xffffbe7a0008  movk   x1, #0xffff, lsl #32
  > 0xffffbe7a000c  add    x1, x1, x0, lsl #2
    0xffffbe7a0010  ldr    w0, [x1]
    0xffffbe7a0014  ret
native process 41276 In:
(gdb) x/4x $x1
0xffffffff110: 0x00000000      0x11111111      0x22222222      0x3333
(gdb)
```

```
fx700-01-05.local /home/kawakami/mkldnn/xbyak_aarch64_fujitsu/sample% ./table.exe
table[2] = 0x22222222
```

Linaro Connect

Generating and Referencing Table

```
#include <xbyak_aarch64/xbyak_aarch64.h>
using namespace Xbyak_aarch64;
class Generator : public CodeGenerator {
public:
  Generator() {
    Label table_label;
    adr(x1, table_label);
    add(x1, x1, x0, LSL, 2);
    ldr(w0, ptr(x1));
    ret();
    L(table_label);
    dd(0x0);
    dd(0x11111111);
    dd(0x22222222);
    dd(0x33333333);
  }
};
int main() {

  Generator gen;
  gen.ready();
  auto f = gen.getCode<uint32_t (*)(uint32_t)>();
  uint32_t a = 2;
  printf("table[%d] = 0x%08x\n", a, f(a));
}
```

JIT-ed code

```
B+  0xffffbe7a0000  adr    x1, 0xffffbe7a0010
  > 0xffffbe7a0004  add    x1, x1, x0, lsl #2
    0xffffbe7a0008  ldr    w0, [x1]
    0xffffbe7a000c  ret
    0xffffbe7a0010  .inst  0x00000000 ; undefined
    0xffffbe7a0014  add    w17, w8, #0x444
    0xffffbe7a0018  .inst  0x22222222 ; undefined
(gdb) x/4x $x1
0xffffbe7a0010: 0x00000000      0x11111111      0x22222222      0x33
(gdb)
```

```
fx700-01-05.local /home/kawakami/mkldnn/xbyak_aarch64_fujitsu/sample% ./table_gen.exe
table[2] = 0x22222222
```

Precautions1

- Xbyak_aarch64 can output instructions that cannot be executed on the CPU running Xbyak_aarch64.
  - Your CPU may not have support for cryptographic, atomic, SVE instructions etc., but Xbyak_aarch64 running on your CPU can output machine code of these instructions, which raises the illegal instruction exception.
    -> Please check your CPU capability and chose the mnemonic functions.
  - In an extreme case, if Xbyak_aarch64 is run on an x64 machine, any ARMv8-A machine code generated by Xbyak_aarch64 causes the exception.

Precautions2

- Xbyak_aarch64 does not validate every argument passed to the mnemonic functions.
  - Example: immediate value of FMOV
    FMOV copies an immediate floating-point constant into every element of SIMD&FP register
    FMOV <Vd>.<T>, **#<imm>**

Only these constant values are allowed for **#<imm> of "FMOV"**

From "Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile"

Table C2-2 Floating-point constant values

efgh \ bcd	000	001	010	011	100	101	110	111
0000	2.0	4.0	8.0	16.0	0.125	0.25	0.5	1.0
0001	2.125	4.25	8.5	17.0	0.1328125	0.265625	0.53125	1.0625
0010	2.25	4.5	9.0	18.0	0.140625	0.28125	0.5625	1.125
0011	2.375	4.75	9.5	19.0	0.1484375	0.296875	0.59375	1.1875
0100	2.5	5.0	10.0	20.0	0.15625	0.3125	0.625	1.25
0101	2.625	5.25	10.5	21.0	0.1640625	0.328125	0.65625	1.3125
0110	2.75	5.5	11.0	22.0	0.171875	0.34375	0.6875	1.375
0111	2.875	5.75	11.5	23.0	0.1796875	0.359375	0.71875	1.4375
1000	3.0	6.0	12.0	24.0	0.1875	0.375	0.75	1.5
1001	3.125	6.25	12.5	25.0	0.1953125	0.390625	0.78125	1.5625
1010	3.25	6.5	13.0	26.0	0.203125	0.40625	0.8125	1.625
1011	3.375	6.75	13.5	27.0	0.2109375	0.421875	0.84375	1.6875
1100	3.5	7.0	14.0	28.0	0.21875	0.4375	0.875	1.75
1101	3.625	7.25	14.5	29.0	0.2265625	0.453125	0.90625	1.8125
1110	3.75	7.5	15.0	30.0	0.234375	0.46875	0.9375	1.875
1111	3.875	7.75	15.5	31.0	0.2421875	0.484375	0.96875	1.9375

Precautions2

- Xbyak_aarch64 does not validate every argument passed to the mnemonic functions.
  - Example: immediate value of FMOV

```
Generator() {
  fmov(v0.s, 3.0);
  fmov(v0.s, 3.33);
```

JIT-ed code
3.0 is OK
3.33 is NG

```
0xffffbe7a0000   fmov    v0.4s, #3.0000000000000000e+00
0xffffbe7a0004   fmov    v0.4s, #3.2500000000000000e+00
```

If you use values that are not listed in the table,
the operation of the mnemonic function is undefined.
The operand validation is the future work.

Table C2-2 Floating-point constant values

efgh \ bcd	000	001	010	011	100	101	110	111
0000	2.0	4.0	8.0	16.0	0.125	0.25	0.5	1.0
0001	2.125	4.25	8.5	17.0	0.1328125	0.265625	0.53125	1.0625
0010	2.25	4.5	9.0	18.0	0.140625	0.28125	0.5625	1.125
0011	2.375	4.75	9.5	19.0	0.1484375	0.296875	0.59375	1.1875
0100	2.5	5.0	10.0	20.0	0.15625	0.3125	0.625	1.25
0101	2.625	5.25	10.5	21.0	0.1640625	0.328125	0.65625	1.3125
0110	2.75	5.5	11.0	22.0	0.171875	0.34375	0.6875	1.375
0111	2.875	5.75	11.5	23.0	0.1796875	0.359375	0.71875	1.4375
1000	3.0	6.0	12.0	24.0	0.1875	0.375	0.75	1.5
1001	3.125	6.25	12.5	25.0	0.1953125	0.390625	0.78125	1.5625
1010	3.25	6.5	13.0	26.0	0.203125	0.40625	0.8125	1.625
1011	3.375	6.75	13.5	27.0	0.2109375	0.421875	0.84375	1.6875
1100	3.5	7.0	14.0	28.0	0.21875	0.4375	0.875	1.75
1101	3.625	7.25	14.5	29.0	0.2265625	0.453125	0.90625	1.8125
1110	3.75	7.5	15.0	30.0	0.234375	0.46875	0.9375	1.875
1111	3.875	7.75	15.5	31.0	0.2421875	0.484375	0.96875	1.9375

How to debug programs implemented with Xbyak_aarch64

Debug JIT-ed Code

- So far, there is no efficient way to debug JIT-ed code ☹
- Basically, it's the same as debugging assembler.
  - I often use GDB with "asm" layout.
- JIT-ed code can be dump as a file and disassembled by "objdump".

Debug by GDB

add.cpp

```
16
17
18
19
20          Generator() {
21            add(w0, w0, w1);
22            ret();
23          }
24        };
25        int main() {
B+ 26       Generator gen;
27          gen.ready();
28          auto f = gen.getCode<int (*)(int, int)>();
29          int a = 3;
30          int b = 4;
 > 31       printf("%d + %d = %d\n", a, b, f(a, b));
32        }
```

```
native process 38684 In: main
(gdb) r
Starting program: /home/kawakami/mkldnn/xbyak_aarch64_fujitsu/sample/add.exe

Breakpoint 3, main () at add.cpp:26
(gdb) n
(gdb) p/x f
$1 = 0xffffbe7a0000
(gdb) b *f
Breakpoint 4 at 0xffffbe7a0000
```

1) Set a break point to the address of the function pointer *f*, before it is called.

Debug by GDB

```
Register group: general
x0             0x3                 3                   x1             0x4                 4                   x2             0xffffbe7a0000
x3             0x100               256                 x4             0x100               256                 x5             0x6
x6
x9
x12                                                                                                                                e06a8
x15
x18            0x2000000005        137438953477        x19            0xfffffffffb680     281474976691840     x20            0xa00000080
x21            0x445948            4479304             x22            0x8                 8                   x23            0x1c00000008
x24            0x1400000008        85899345928         x25            0x1200000080        77309411456         x26            0x1e00000080
x27            0x1e00000020        128849018912        x28            0x1200000020        77309411360         x29            0xfffffffffb610
x30            0x4045d4            4212180             sp             0xfffffffffb610     0xfffffffffb610     pc             0xffffbe7a0000
cpsr           0x80000000          [ EL=0 N ]          fpsr           0x0                 0                   fpcr           0x0
vg             0x8                 8

B+> 0xffffbe7a0000  add    w0, w0, w1
    0xffffbe7a0004  ret
    0xffffbe7a0008  .inst  0x00000000 ; undefined
    0xffffbe7a000c
native process 38684
Starting program: /h

Breakpoint 3, main () at add.cpp:26
(gdb) n
(gdb) p/x f
$1 = 0xffffbe7a0000
(gdb) b *f
Breakpoint 4 at 0xffffbe7a0000
(gdb) layout asm
(gdb) c
Continuing.

Breakpoint 4, 0x0000ffffbe7a0000 in ?? ()
(gdb) layout regs
(gdb)
```

4) The program breaks at the start of JIT-ed code

5) Then, you can step through the instruction level with GDB command “si”.

2) Set layout to “asm” or “regs”

3) Continue execution

Dumping JIT-ed Code

```
#include <xbyak_aarch64/xbyak_aarch64.h>
using namespace Xbyak_aarch64;
class Generator : public CodeGenerator {
public:
  Generator() {
    add(w0, w0, w1);
    ret();
  }
};
int main() {
  Generator gen;
  gen.ready();
  auto f = gen.getCode<int (*)(int, int)>();
  int a = 3;
  int b = 4;

  FILE *fp = fopen("dump.bin", "wb+");
  fwrite(gen.getCode(), gen.getSize(), 1, fp);
  fclose(fp);

  printf("%d + %d = %d\n", a, b, f(a, b));
}
```

```
[kawakami@fx700-01-05 sample]$ objdump -D -b binary -m AArch64 dump.bin

dump.bin:     file format binary


Disassembly of section .data:

0000000000000000 <.data>:
   0:   0b010000        add     w0, w0, w1
   4:   d65f03c0        ret
[kawakami@fx700-01-05 sample]$
```

Summary

Summary

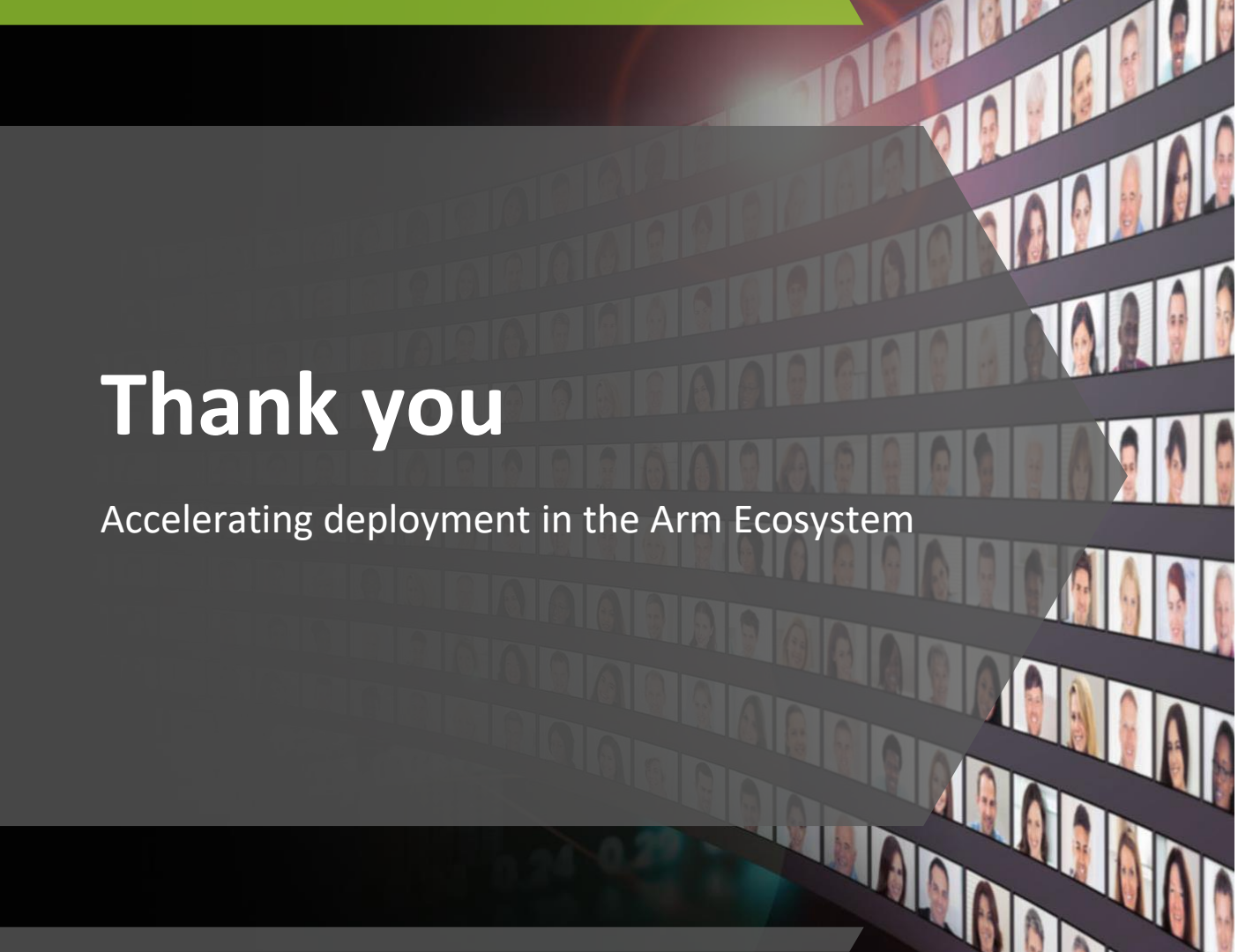
- **Xbyak_aarch64; just-in-time assembler for ARMv8-A + SVE, is introduced,**
  - which can dynamically generate optimized code considering runtime parameters and make it easier than the existing assembler to implement optimized code at the instruction level.
- Xbyak_aarch64 is mainly developed to implement the deep learning processing software on the supercomputer Fugaku, but it can be expected to work with a variety of software for ARMv8-a architecture systems.
- **Xbyak_aarch64 is being developed as an OSS. I hope that many people will use Xbyak_aarch64 on various platforms and participate in its development.**
  - Questions, bug reports, pull requests, etc. on Github are welcome. https://github.com/fujitsu/xbyak_aarch64

Acknowledgment

- The authors thank S. Mitsunari (Cybozu Labs, Inc.), the developer of the original Xbyak. He contributed helpful advice to Xbyak_aarch64 and brushed up the source code.

References

- Xbyak_aarch64: Just-In-Time assembler for ARMv8-A + SVE
  - https://github.com/fujitsu/xbyak_aarch64
- Xbyak: Just-In-Time assembler for x86_64
  - https://github.com/herumi/xbyak
- oneDNN: Deep Learning Processing Library
  - https://github.com/oneapi-src/oneDNN
- oneDNN for A64FX: Deep Learning Processing Library for A64FX
  - https://github.com/fujitsu/oneDNN
- A64FX: CPU designed for high-performance computing and complies with the ARMv8-A architecture profile and the Scalable Vector Extension (SVE)
  - Toshio Yoshida, “Fujitsu High Performance CPU for the Post-K Computer,” in Proc. Hot Chips 30, Aug. 2018.
- “Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile”
- “Procedure Call Standard for the Arm 64-bit Architecture (AArch64)”
- **TechBlog**
  - https://blog.fltech.dev/entry/2020/11/19/fugaku-onednn-deep-dive-en
  - https://blog.fltech.dev/entry/2020/11/18/fugaku-onednn-deep-dive-ja (in Japanese)



Thank you

Accelerating deployment in the Arm Ecosystem

Linaro Connect
VIRTUAL • SPRING 2021

What is Xbyak_aarch64?

- Xbyak_aarch64 (https://github.com/fujitsu/xbyak_aarch64) is the Just-In-Time (JIT) assembler for ARMv8-A + Scalable Vector Extension (SVE), inheriting the concept of Xbyak, written in C++11.
- Xbyak (https://github.com/herumi/xbyak) is the Just-In-Time assembler for x86_64 instruction set architecture (ISA),
  - developed by S. Mitsunari (Cybozu Labs, Inc.),
  - pronounced “kəi-bja-k” (I'm not sure the correct spelling by IPA), https://translate.google.com/?hl=ja&sl=ja&tl=en&text=kaibyaku
  - The word “Xbyak” is derived from Japanese word “開闢”.
    - Its meanings is "the beginning of the world", "exploring the unexplored", etc.
- The main purpose of developing Xbyak_aarch64 is to port oneDNN, a deep learning processing library for x86_64, to A64FX (ARMv8-A + SVE).