Current state of GCC support for PPU

Arthur Heimbrecht

November 21, 2016

Basic compiler structure

Front-end

- recognizes language
- type checking
- pre-processing
- genreates Immediate Representation

Middle-end

- general optimizations to IR
- "compiler magic"

Back-end

- target specific
- further/final optimization
- register allocation (spilling)
- memory handling

Compiler backend

- target files: target.h, target.md, target.c
- important "variables"
  - register_types, _numbers, _names
  - wordsize
  - basic insns
  - constraints ("r", "m"...)
- PPU characteristics:
  - POWER architecture
  - general and vector registers (32 each)
  - synram

PPU programming until now

- binutils patch + fxv.h
  - close to actual assembly
  - efficient execution
- not user-friendly
- reaccuring code

code

```
uint32_t v1, v2, v3;
fxvsplatb (1,1);
fxvstore (&v1, 1);
fxvsplatb (2,2);
fxvstore (&v2, 2);
fxvadd  (0,1,2);
fxvstore (&v3, 0);
```

machine instructions

```
24: mr        r31,r1
28: li        r9,257
2c: fxvsplath 1,r9
30: addi      r9,r31,8
34: fxvstax   1,0,r9
38: li        r9,514
3c: fxvsplath 2,r9
40: addi      r9,r31,12
44: fxvstax   2,0,r9
48: fxvaddbm  0,1,2
4c: addi      r9,r31,16
50: fxvstax   0,0,r9
```

GCC vs. LLVM

- GCC already working
- better kown
- small internal changes between versions $\rightarrow$ less maintenance

Current state of PPU backend

First steps

- start with rs/6000 back-end
- add header files and command line option -ms2pp
- add s2pp register type $\rightarrow$ overloaded float regs
  - needed own internal vector type, bit-masks,...
  - AltiVec as blueprint
  - a lot of trouble
- basic insns
- support vector type and built-ins
- implement "helper functions"

Current state of PPU backend

Create built-in function in 3,5 steps

1. s2pp.md
    - create insn in RTL
2. rs6000-builtin.c
    - define built-in name
    - connect with insn
3. rs6000-c.c
    - set output/input type
    - built-in already works
4. s2pp.h
    - define built-in aliases
    - suggestions for name convention?

Current state of PPU backend

Code comparison

old code

```
uint32_t v1, v2, v3;
fxvsplatb (1,1);
fxvstore (&v1, 1);
fxvsplatb (2,2);
fxvstore (&v2, 2);
fxvadd  (0,1,2);
fxvstore (&v3, 0);
```

old assembly code

```
24: mr         r31,r1
28: li         r9,257
2c: fxvsplath  1,r9
30: addi       r9,r31,8
34: fxvstax    1,0,r9
38: li         r9,514
3c: fxvsplath  2,r9
40: addi       r9,r31,12
44: fxvstax    2,0,r9
48: fxvaddbm   0,1,2
4c: addi       r9,r31,16
50: fxvstax    0,0,r9
```

new code

```
vector unsigned char
v1, v2, v3;
v1 = fxv_splat(1);
v2 = fxv_splat(2);
v3 = fxv_add(v1. v2);
```

Current state of PPU backend

Code comparison

old code

```
uint32_t v1, v2, v3;
fxvsplatb (1,1);
fxvstore (&v1, 1);
fxvsplatb (2,2);
fxvstore (&v2, 2);
fxvadd  (0,1,2);
fxvstore (&v3, 0);
```

new assembly code

```
60: mr        r31,r1
64: fxvsplatb 12,r1
68: li        r9,16
6c: fxvstax   12,r31,r9
70: fxvsplatb 12,r2
74: li        r9,32
78: fxvstax   12,r31,r9
80: fxvlax    11,r31,r9
84: li        r9,32
88: fxvlax    12,r31,r9
8c: fxvaddbm  12,11,12
90: li        r9,48
94: fxvstax   12,r31,r9
```

new code

```
vector unsigned char
v1, v2, v3;
v1 = fxv_splat(1);
v2 = fxv_splat(2);
v3 = fxv_add(v1. v2);
```

Conclusion

- partly usable
- add remaining insns and built-ins
- add more complex built-ins? (e.g. multipy and add, scalar multiplication...)
- write manual
- write patch

Questions or Suggestions?