Current state of GCC support for PPU

Arthur Heimbrecht

November 20, 2016

1. Basic compiler structure
2. PPU programming until now
3. Why GCC?
4. Current state of PPU backend

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

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

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 (is more complex)
  - 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

```
asm code
```

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

```
asm code
```

new code

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

Conclusion

- currently 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?