Cray Machine Description.

Toon Moene.

Rationale.

- The Cray architecture was designed in the first half of the 1970s and therefore predates GCC by at least a decade.
- Although it didn’t “predate” the C language, strictly speaking, it wasn’t made to run C programs easily (as I will show shortly).
- In fact, its main purpose was to run nuclear bomb simulations and weather forecasting models, both written in Fortran.
- So – how well can GCC work on this architecture ?
- Don’t try this at home unless you have a permit to use a Freon-cooled device (and have 3-phase, 208 V, 400 Hz, 150 kW power).

Cray architecture – outline (I).

- The Cray architecture is a typical (some would say: archetypical) RISC load/store design.
- Register-to-register operations have three operands.
- Addresses point to 64-bit words in memory. (Note: this talk assumes #define BITS_PER_UNIT 64 will suffice).
- It has eight computational registers (S, 64 bits), which contain either an integer or a floating point number ...
- … and eight address registers (A, 24 bits).
- Both register sets have a 64-member “shadow” register file (S ↔ T and A ↔ B).

Cray architecture – outline (II).

- In addition, there are eight 64-element 64-bit vector registers (V).
- Both S and A registers are loaded from and stored to memory using the address contained in an A register plus a 20-bit offset (using A0 means using only the 20-bit offset).
- Cray does not have a floating point division instruction – you have to use a floating point reciprocal instruction followed by a multiplication and a two step Newton approximation described in the manual.
- Needless to say Cray floating point arithmetic doesn’t follow the IEEE-754 standard because it preceded it by a decade.

Challenges (prelude).

- In the autumn of 1990 one of my previous employers bought a 4 processor Cray Y-MP, running the UNICOS operating system.
- Because it was acquired to be the national academic supercomputer, it had to function well within the Dutch academic internet (IPv4) at that time.
- It turned out rather quickly that I would need ‘traceroute’ for that purpose – which unfortunately didn’t come with the machine.
- As you might imagine, porting a thoroughly 32-bit code to a machine that has only 64-bit arithmetic was a challenge.

Challenges (I).

- So, for fun - I tried to figure out what the C compiler did with:

```
void copy (char *s, char *d)
{
    while (*d++ = *s++)
        ;
}
```

It turned out that character pointers were implemented by stuffing a word address and an (either byte or bit – can't recall) offset into a 64-bit computational register. Character arrays were being stored 8 characters to a word in both memory and registers.

Needless to say, that made the loop body rather "interesting" – I counted 80 (*eighty*) instructions in total. Can GCC do something similar (stuff two "parts" of the character address into a struct) ? Or do we have to store one character per word ? Is this something SLP can help with ?

Challenges (II).

- Earlier, I pointed out that both the eight S registers as well as the eight A registers have a 64-member “shadow” register file.
- Of course, it would be nice if register allocation could be “encouraged” to use those registers (“T” and “B”) as targets for spilling in preference of using the stack.
- In this context it might be useful to mention that blocks of T or B registers can be transferred to or from memory.
- Does LRA have a mechanism for such a “preferred spilling destination” ?

Challenges (III).

- Special care has to be taken for branching.
- One would think that on a 64-bit word addressable machine, branching would be to a word boundary.
- However, instructions can be 1 or 2 parcels (of 16 bits) “wide”, so branching is to a parcel within a word (its offset is encoded in the lower two bits of the branch operand that has the “address”).
- Obviously, all hell breaks lose if you branch into the “middle” of a 2-parcel instruction.

Concluding remarks.

- I haven’t yet looked at the vectorization challenges, but they might be minimal.
- The vector registers are large in comparison to what is used on other architectures (64 elements) – but I don’t think that should pose a problem.
- Cray has both a vector length and a vector mask register.
- For those who want to look further into this are advised to visit the Cray architecture manual
- I plan to do this on a personal branch (if that’s visible to all).

Post Scriptum I: Size Matters.

- Kernighan and Ritchie made it quite clear in their “white book” paragraph on “Data type and sizes”, that
- “[An] int is an integer, typically reflecting the natural size of integers on the host machine.”
- For the Cray architecture, that would mean an int (and hence SImode) would have to be defined as a 64 bit entity.
- Hans-Peter Nilsson expressed to me that he highly doubted that would work in GCC, as there are no instances of machine descriptions with SImode being larger than 32 bits.

Post Scriptum II: Addressing subword entities (a).

- On the Cray architecture, addresses (as expressed via the A registers) refer to the location of 64-bit words.
- So a challenge is to address subword entities. Vladimir Makarov suggested the following to me:
- A solution would be to let Pmode address bits, and add specific define_expand sequences to store or extract subword entities “inside” a 64-bit word.
- This could be done both for memory words, as well as inserting or extracting subwords from (S) registers.

Post Scriptum II: Addressing subword entities (b).

- However, there is a snag, as Richard Earnshaw pointed out to me: Modern C Standards assume (paragraph 3.14 in the 2017 Standard):
- “Two threads of execution can update and access separate memory locations without interfering with each other.”
- It is abundantly clear from the examples given that that means “two consecutive char elements in a struct”, which of course is not going to work on a machine where the unit of memory access/update is a 64-bit word.

Post Scriptum III: Using T and B registers.

- Ideally, S registers would have to be spilled to T registers and A registers to B registers, as that would prevent storing them in memory (on the stack).
- According to Vladimir Makarov (author of the LRA register allocator), this should be possible if the T and B registers could be relegated to separate register classes that are not connected to computational instructions.
- It would, however, still need some extension to the register allocator’s algorithms.