<table>
<thead>
<tr>
<th>Arch</th>
<th>Server</th>
<th>Desktop</th>
<th>Embedded</th>
<th>MCU</th>
<th>DSP</th>
<th>GPU</th>
<th>Hardware</th>
<th>Reg size</th>
</tr>
</thead>
<tbody>
<tr>
<td>AArch64</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>64</td>
</tr>
<tr>
<td>AMDGPU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>ARM</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>32</td>
</tr>
<tr>
<td>BPF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>Hexagon</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>Lanai</td>
<td>?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>?</td>
<td>32</td>
</tr>
<tr>
<td>Mips</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>32, 64</td>
</tr>
<tr>
<td>MSP430</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>16</td>
</tr>
<tr>
<td>NVPTX</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>16, 32, 64</td>
</tr>
<tr>
<td>PowerPC</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>32, 64</td>
</tr>
<tr>
<td>Sparc</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>64</td>
</tr>
<tr>
<td>SystemZ</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>32, 64</td>
</tr>
<tr>
<td>WebAsm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>32, 64</td>
</tr>
<tr>
<td>X86</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>16, 32, 64</td>
</tr>
<tr>
<td>XCore</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>32</td>
</tr>
<tr>
<td>AVR</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>8</td>
</tr>
<tr>
<td>RISC-V</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>32</td>
</tr>
</tbody>
</table>
Out-of-tree Architectures

- MCUs, DSPs
- 8/16/24/40 bit registers, >8 bit bytes
- Novel addressing modes
- Complex register constraints
- Size > performance
- Can't always upstream
  - Exposes confidential hardware details
  - Only internal use
  - Lots of generic changes
• Community driven, incrementally update architecture to solve community needs
• Aim to be in-tree to avoid bit-rot
• Smooth the path for other back-ends to come in-tree
• RISC with MCU and DSP features
AAP Design

4 - 64 Registers

Code Memories (words)

Data Memories (bytes)

ALU

STATUS/PC
AAP exists to improve LLVM. Some areas we’re looking at:

- Non-octet chars
- Non-power of 2 registers
- Multiple function pointer sizes
- Finer-grained control over legalization
- Stack cheaper than registers
Non-octet Chars

Very common in DSPs, but 8-bit chars are hardcoded in a few places in LLVM/Clang

- `getPointerSizeInBits`
- `getTypeStoreSize/getTypeStoreSizeInBits`
- ....

Has appeared a couple of times in the mailing list (including /w patches). But patches never made it in.
Also very common in DSPs. Generic LLVM only has power-of-2 in the MVT.

Support for common DSP register sizes would be nice (i24)

Again, patches have appeared on the mailing list. Understandably, with no in-tree targets the patches weren't included.
Some architectures have code which lives in different address spaces.

Function pointers may be different sizes

- And different size pointers may point to the same function

Call and return instructions used may differ depending on pointer type

Can expand meaning of address_space in LLVM to work for functions.
Finer-grained legalization control

- Some architectures have multiple register sizes
- May only be able to do a subset of operations on the larger register
- But if the larger register is legal, then LLVM will use it liberally.
- Can occur if your address registers are larger than your data registers.
Some architectures can perform operations on the stack directly.

May be cheaper than registers:
- For a subset of stack locations
- For a subset of operations

Want to sometimes treat stack slots like registers.

```
load   r0,   sp+2
add    r0,   r1
store  sp+2, r0

add    sp+2, r1
```
Stack cheaper than registers

- Cost of spilling a register varies
  - Free if subsequent uses of the value can operate directly on the stack
- Stack slot allocation varies
  - Allocate cheap stack slots to minimize number of loads into registers
- Instruction selection varies
  - Choose instructions to prefer operations which can use the stack
D23665 – ELF Definitions
D23666 – Backend Stub
D23667 – Tablegen
D23771 – MC
D23772 – AsmParser
D23773 – InstPrinter