**Das kann die ALU:**

* Zwei Register miteinander Addieren (ADD), es kann ein vorher ermittelter Überlauf (das Carry aus dem SREG) miteinbezogen werden (ADC).
* Analog dazu zwei Register subtrahieren, ggf. auch hier mit Carry (SUB bzw. SBC)
* Von einem Register eine Konstante abziehen, auch hier ggf. mit Carry (SUBI, SBCI)
* Auf ein Doppelregister (16bit) eine Konstante addieren (ADIW) bzw. von dort subtrahieren (SBIW)
* Zwei Register bitweise logisch verknüpfen (AND, OR, EOR)
* Ein Register mit einer Konstante bitweise logisch verANDen (ANDI) bzw. verORen (ORI)
* Von einem Register das Einer- (COM) bzw. Zweierkomplement (NEG) bilden
* Ein Register inkrementieren (INC) bzw. dekrementieren (DEC)
* Zwei Register multiplizieren, und zwar beide unsigned (MUL), beide signed (MULS) oder eins singned und eines unsigned (MULSU), analog dazu fraktionelle Multiplikation (FMUL, FMULS, FMULSU)

**Hierher gehören eigentlich auch noch folgende Bit-Funktionen, die ALU kann:**

* Die Bits eines Registers logisch nach links (LSL) oder rechts (LSR) schieben, dabei rutscht eine null auf die leere Stelle, das rausgeschobene Bit fällt ins Carry
* Statt der null kann auch das Carry auf die leere stelle gerollt werden (ROL bzw. ROR)
* Die Bits eines Registers arithmetisch nach rechts schieben (ASR, Integer Division durch zwei)

**Und der Vollständigkeit halber noch die drei Vergleiche - das sind in Wirklichkeit nur Subtraktionen ohne Zurückschreiben des Ergebnisses. Trotzdem wird das SREG manipuliert:**

* CP vergleicht (subtrahiert) zwei Register
* CPC subtrahiert zusätzlich das Carry
* CPI subtrahiert eine Konstante vom Register

Streng genommen haben wir damit alle Instruktionen, in denen die ALU durch eine Operation Bits im SREG manipuliert.  
(hinter SBR verbirgt sich in Wirklichkeit ORI, hinter CBR ANDI, CLR ist eigentlich ein EOR, SER ein LDI (also gar keine Rechenoperation, deswegen kommt auch das Zero nicht), TST ist ein AND)

**Verzweigungen. Es gibt unbedingte:**

* RJMP springt relativ von der derzeitigen Position an eine feste Adresse (12Bit) (\*)
* JMP springt absolut auf eine feste Adresse (22Bit Adressraum)
* IJMP verwendet den Inhalt des Z-Doppelregisters (16Bit Adressraum) als Sprungziel
* EIJMP stellt dem Z noch ein drittes Register zur Seite (22Bit Adressraum)
* RCALL(\*), CALL, ICALL und EICALL machen dasselbe, legen lediglich vorher eine
* Rücksprungadresse auf den Stack -> Subroutine Calls eben
* RET verwendet die obersten Bytes des Stack als Spungziel (Subroutine Return), RETI setzt
* nebenbei das I-Flag (aber nicht bei XMEGAS)

**Und bedingte..  
Bedingung ist hier immer ein Bit des SREG (siehe oben)**

* BRBS springt, wenn das Bit gesetzt ist
* BRBC springt, wenn es gelöscht ist
* Natürlich gibt es für jedes Bit je für die beiden Fälle eigene Mnemonics (beim Carry sogar doppelt) - aber vom Maschinencode her ist es BRBS/BRBC
* Und es gibt Skips - unter bestimmten Bedingungen kann der nächste Befehl übersprungen werden:
* CPSE vergleicht zwei Rechenregister, und "skippt" wenn gleich
* SBRC und SBRS prüfen ein Bit in einem Rechenregister, und skippen wenn dieses gelöscht/gesetzt ist
* SBIC und SBIS machen dasselbe mit einem Bit eines (der ersten 32) I/O-Register
* (\*) RJMP und RCALL sind eigentlich relative Sprünge - ein vorzeichenbehafteter 12Bit-Integer legt das jeweilige Ziel ausgehend von der aktuellen Position fest. Folgt man den derzeitigen Versionen des Studios könnte man also mit RJMP -10 zurückspringen, oder mit RCALL 42 vorwärts. Der entsprechende Opcode im AVR würde sogar gehen, ABER das Atmel Studio setzt den nicht so um.

**Für den Zugriff auf das SRAM (wobei hier meist die Rechenregister adresstechnisch mit eingebunden sind (manchmal sogar ein Abbild der nichtflüchtigen Speicher) gibt es:**

* **Lds/Sds** lädt vom/speichert direkt auf die angegebene SRAM-Adresse
* **Ld/St** lädt vom/speichert indirekt auf die, durch eines der Doppelregister X,Y oder Z referenzierte SRAM-Adresse. Dabei kann außerdem der Pointer Pre-decrementiert oder Post-inkrementiert werden.
* **Ldd/Std** nutzt analog das Y oder Z als Pointer, zusätzlich wird ein konstanter Offset (6Bit) dazuaddiert. Zusammen mit LD/ST kann man so geschickt mehrere Listen/Arrays/… gleichzeitig verarbeiten, ohne den Pointer umladen zu müssen. Insbesondere in Schleifen...
* **Push** kopiert den Inhalt eines Rechenregisters ins SRAM, adressiert durch den Stackpointer (welcher dabei automatisch dekrementiert), **Pop** macht das Gegenteil davon.
* Den XMegas stehen außerdem noch **Las**, **Lac**, **Lat** und **Xch** zur Verfügung. Damit können Bits in SRAM-Registern direkt gesetzt/gelöscht/getoggelt werden. Hintergrund ist, daß dort fast alle Peripherals außerhalb des konventionellen I/O-Space liegen. In und Out greifen also nicht, Sbi/Cbi/Sbic/Sbis erst recht nicht.
* Für den Zugriff auf den Flash gibt es (**E**)**Lpm**und **Spm**, referenziert wird die Adresse durch das Z-Doppelregister.

**Bei den Rechenregistern selbst:**

* kann mit **Ldi** eine Konstante geleden werden (welche quasi mit dem Ldi im Flash abgelegt ist)
* mit **Mov** der Inhalt eines Registers in ein anderes kopiert werden
* mit **Movw** dasselbe mit je zwei Registern (also 16Bit)
* kopiert **BST** ein Bit aus einem Rechenregister in's T-Flag des SREG, **Bld** lädt dieses Flag in ein Rechenregister-Bit
* Bei den Bit und Bit-Test Instruktionen fehlen noch:
* **Swap** - tauscht die Nibble eines Rechenregisters
* **Sbi**/**Cbi** - setzt/löscht ein Bit eines I/O-Registers direkt - ohne Umweg über ein Rechenregister (nur die ersten 0x1F I/O-Register sind adressierbar)
* **Bset**/**Bclr** - setzt/löscht direkt ein Bit im SREG (hier gibt's auch wieder für jedes Bit zwei separate Mnemonics, klar)
* Zur "Steuerung" der MCU selbst gibt es dann noch folgende Instruktionen:
* **Nop** - macht einen Takt lang nichts
* **Sleep** - aktiviert den, im entsprechenden I/O-Register ausgewählten Sleep-Mode
* **Wdr** - setzt den Watchdog-Counter zurück
* **Break** -ermöglicht einem Debugger Zugriff auf den Controller